ALICE Expert System

C Ionita[1] **and F Carena**[2]

[1,2] ALICE DAQ, CERN, Route de Meyrin 385, 1217 Meyrin, Switzerland

E-mail: `costin.ionita@cern.ch`

Abstract. The ALICE experiment at CERN employs a number of human operators (shifters), who have to make sure that the experiment is always in a state compatible with taking Physics data. Given the complexity of the system and the myriad of errors that can arise, this is not always a trivial task. The aim of this paper is to describe an expert system that is capable of assisting human shifters in the ALICE control room. The system diagnoses potential issues and attempts to make smart recommendations for troubleshooting. At its core, a Prolog engine infers whether a Physics or a technical run can be started based on the current state of the underlying sub-systems. A separate C++ component queries certain SMI objects and stores their state as facts in a Prolog knowledge base. By mining the data stored in different system logs, the expert system can also diagnose errors arising during a run. Currently the system is used by the on-call experts for faster response times, but we expect it to be adopted as a standard tool by regular shifters during the next data taking period.

1. Introduction

The ALICE collaboration is studying a state of matter called quark-gluon plasma, which existed just after the Big Bang when the Universe was still extremely hot. In order to achieve this goal, many different detectors have been put in place, each providing a different piece of information to physicists. The control of these detectors is achieved using several "online systems" [1]: Detector Control System (DCS), Data Acquisition (DAQ), Trigger system (TRG), and High Level Trigger (HLT). On top of these, the Experiment Control System (ECS) coordinates the actions performed by all online systems.

Keeping these systems in a state compatible with Physics data taking is not always a trivial task. Alongside hardware or software faults, human error is also a major factor in lowering the data taking efficiency. This comes as a result of the fact that the shifters who monitor the state of these systems are not always expert users. For this reason, shifters often do not know how to handle the inherent complexities and must seek further assistance from an on-call expert, usually resulting in lost data taking time.

After analyzing the most frequent problems occuring in practice, we came to the conclusion that not all problems require advanced expert knowledge. In fact, some of these issues can be diagnosed, and then solved, just by inspecting the state of certain sub-systems. In this paper, we describe the ALICE expert system, which aims to automate the diagnosis of frequently occuring issues and make smart recommendations for troubleshooting.

In order to understand the state of the ECS at any given moment, the expert system queries a number of SMI objects [2], which describe the behaviour of the ECS as a finite state machine. This information is correlated with the information extracted from the configuration databases, and then stored as facts in a Prolog knowledge base. Based on the facts known at a certain time, the system infers whether a Physics or technical run is possible. If a run cannot be started, the system prompts the user with a set of failure reasons, as well as with possible solutions provided by human experts.

Note however that some problems cannot be detected a priori. In this case, the Physics run can actually start, but it will fail at run time. In this scenario, the system looks at the error messages (with the highest severity) stored by the ALICE infoLogger system [3], and attempts to make a recommendation using again human expert knowledge.

2. Architectural overview

Before going into more details, it is important to get a high level overview of the data flow in the ALICE Data Acquisition (DAQ) system. Every collision produces a very large amount of data, recorded as "event fragments" using dedicated electronic equipment. Event fragments are sent from the Front End Read Out (FERO) electronics of the detectors, through optical links, to Local Data Concentrators (LDC), which generate a number of sub-events. Each sub-event is then shipped through a Gigabit Ethernet network to Global Data Concentrators (GDC) where the full events are built. All LDCs and GDCs used in a given run are grouped together under a high level entity called a partition.

The expert system builds its knowledge base using a C++ component that extracts information from multiple sources. First, we have the configuration databases containing vital information, such as the list of detectors included in a partition, whether the GDCs are selected statically or dynamically, and so on. The *DataExtractor* connects to these databases, extracts and parses the raw data, and then stores it in more convenient data structures. At the next stage, the *KBManager* takes the information received from the *DataExtractor*, aggregates it with state information provided by the different SMI objects, and then asserts it as Prolog facts (Fig. 1). The exact SMI objects that have to be queried in this process have been established using human expert knowledge of the ECS.

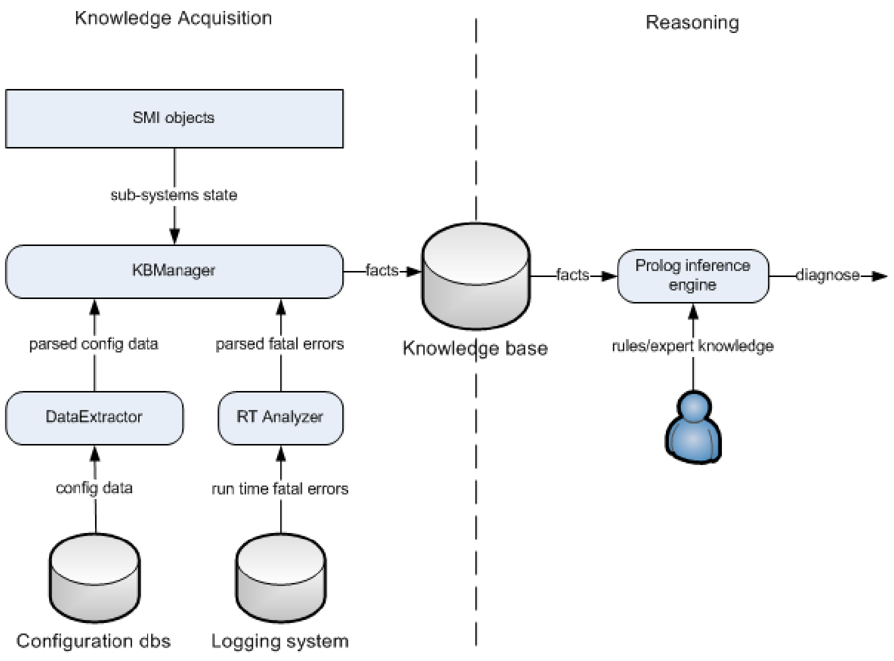


Figure 1. Architectural overview

Another source of information is the infoLogger system, which provides an overview of the errors occuring at run time. The Run Time Analyzer (*RT Analyzer*) connects to this database and attempts to extract the errors with the highest severity, which are usually the root cause of a run interruption. Similar to the *DataExtractor*, these errors are again parsed and put into a more convenient format, while at the next stage, the *KBManager* asserts them as Prolog facts into our knowledge base.

It is important to note that the reasoning is completely decoupled from the knowledge acquisition. Thus, the *KBManager* performs no filtering of the data, as its only role is to build a large pool of facts that can be used for reasoning. This means that the process can be inefficient at times, since some of the asserted facts might be useless. For instance, there is no need to store the information of all possible GDCs if the GDCs are selected statically. Ideally we would only store information about the required GDCs. However, the *KBManager* is not aware whether the GDCs are selected dynamically or statically and it does not attempt to build any decision trees. This is done at a later stage by the inference engine. The reason we have chosen this design is to keep the reasoning contained within the Prolog inference engine, and not spread across other components. If the logic changes in the future, we need only modify the predicate definition in the inference engine, while the knowledge acquisition component will be kept the same.

3. Knowledge representation and reasoning

A generic expert system is made up of three main components, as described in Fig. 2: the production memory, the working memory and the inference engine.

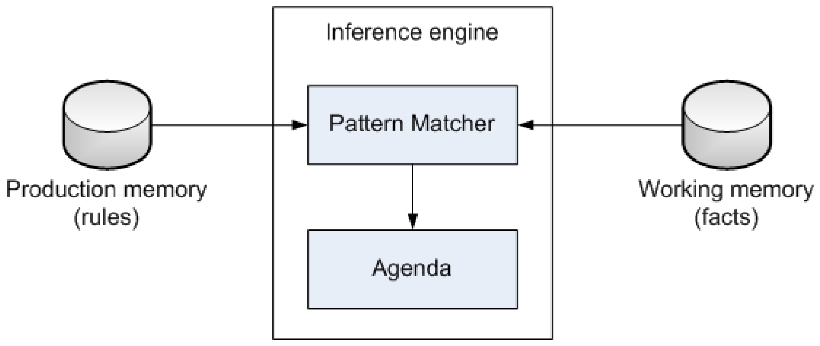


Figure 2. Generic expert system

The working memory contains all the known facts about the world. For instance, if detector *a* is running, then we can represent this information as a predicate *running(a)*. The production memory contains all the rules (expert knowledge) that the system uses to assert new facts or to determine the satisfiability of a goal. In the example below, a control server is available if it is running and not locked:

```
rc_servers_available([H|T]) :-
  check_running(H),
  check_not_locked(H),!,
  rc_servers_available(T).
```

The inference engine can reason about the facts in the knowledge base using either forward chaining or backward chaining. In the first approach, the engine runs through all the rule definitions and triggers rules whose conditions match facts in the knowledge base. These rules add new data to the knowledge base, which can then trigger more rules. The process continues iteratively until no rules are triggered anymore (Fig. 3).

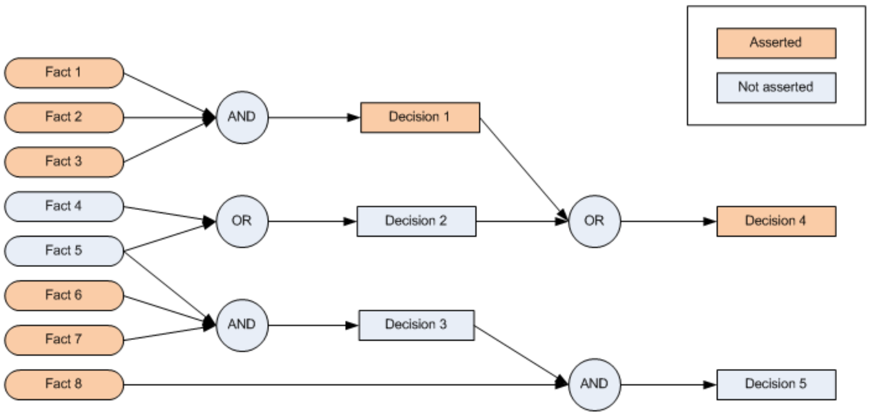


Figure 3. Forward chaining

In backward chaining, the system starts from a goal definition and attempts to satisfy its preconditions. The process continues recursively until a predicate can be evaluated as true or false. These predicates correspond to the leaves of a decision tree (Fig. 4). Note that in general a closed world assumption is required, i.e. if a predicate is not explicitly indicated to be true, then it can be assumed to be false.

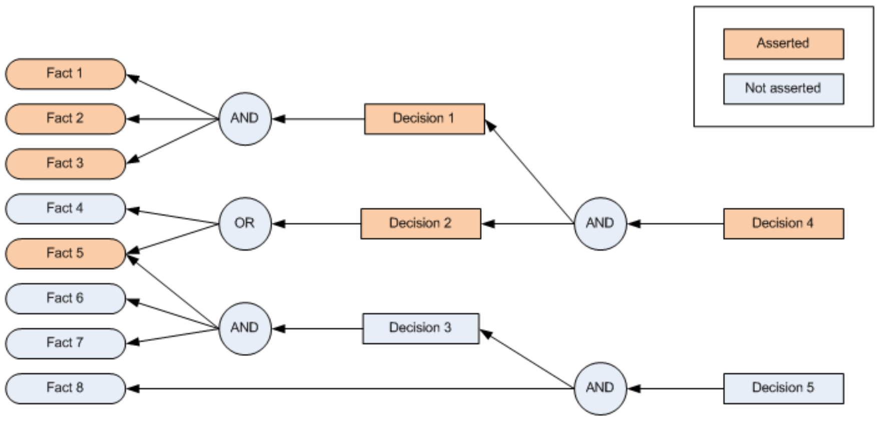


Figure 4. Backward chaining

The main disadvantage of forward chaining is that we cannot establish the reasons why a goal could not be satisfied, which is a critical information when assisting the human shifters. Backward chaining will attempt to backtrack whenever a predicate cannot be satisfied, and therefore, we can use this feature to store all reasons for which a certain action could not be performed. Consequently, Prolog is a natural choice for our inference engine, since it uses backward chaining.

Note, however, that Prolog cannot deal natively with the requirement of providing all reasons for a failed predicate. For this reason, we had to use a workaround, taking advantage of its backtracking capabilities. Each basic fact is wrapped with a top level predicate, that adds a reason to a list whenever the basic fact is not true, as shown below:

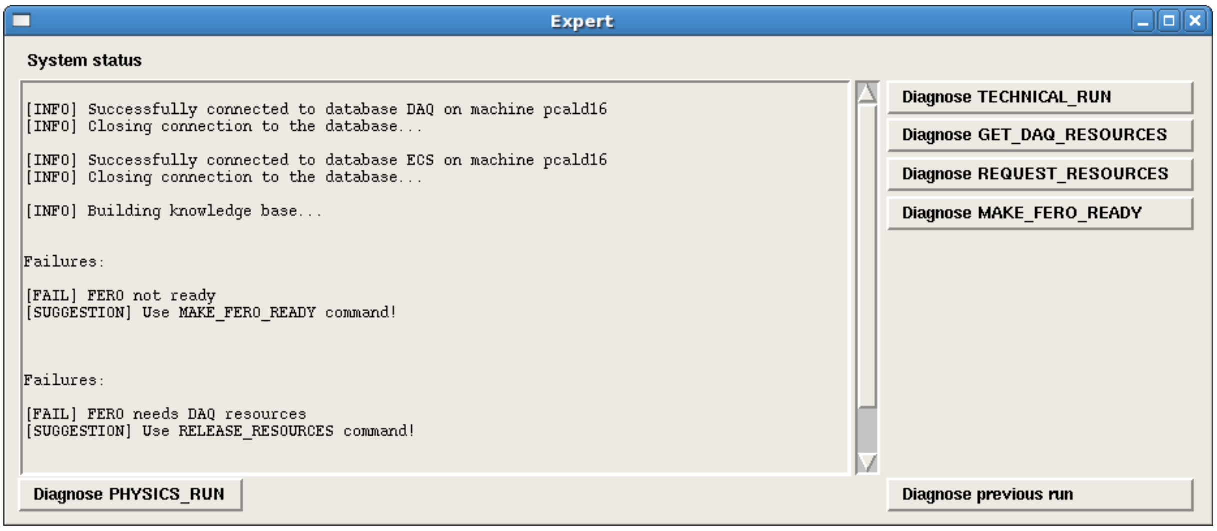
```
check_granted(X) :-
  granted(X);  %if true, continue

  failures(F), % if not true, get the list of current failures
  append(F,[X,'not granted'],F2), % append new failure
  retractall(failures(_)),% remove old list
  assert(failures(F2)), fail. % assert new list and return false
```

4. A priori and a posteriori detection

The main role of the system is to assist the human shifters in determining why Physics data taking is not possible in a given system state or configuration. Apart from that, an additional set of actions (pre-requisites for Physics data taking) can also be diagnosed individually. The entire process is fully automatic, and the system is able to recursively drill down on a certain problem, until the most specific diagnosis is found. As shown in Fig. 5, the Physics run cannot start because the FERO is not ready. While in principle this problem can be easily solved by running a MAKE_FERO_READY command, the system continues the analysis and determines that such a command would fail since the DAQ resources are blocked by another entity. Thus, the actual recommendation given by the system is to first release the DAQ resources, which, in turn, allows both the FERO ready and the starting of a Physics run.

Another important observation is that some of the facts cannot be known a priori and the potential problems can only be detected at run time. This is why we added the *RT Analyzer* as a separate component. Unlike a priori diagnosis, run-time analysis is not real-time, as the system needs to wait until the events have been stored by the logging system. Note that this does not interfere in any way with the typical workflow, as no action can be performed anyway until a Physics run comes to a complete halt.



```
[INFO] Successfully connected to database DAQ on machine pcald16
[INFO] Closing connection to the database...

[INFO] Successfully connected to database ECS on machine pcald16
[INFO] Closing connection to the database...

[INFO] Building knowledge base...


Failures:

[FAIL] FERO not ready
[SUGGESTION] Use MAKE_FERO_READY command!


Failures:

[FAIL] FERO needs DAQ resources
[SUGGESTION] Use RELEASE_RESOURCES command!
```

Figure 5. Tcl/Tk GUI

5. Conclusions

In this paper, we have described an expert system capable of assisting human shifters in the ALICE control room. Our system extracts information from the configuration databases and the logging system, and then dynamically builds a Prolog knowledge base. The rules used for inference have been designed using human expert knowledge.

Our next step is to better integrate the expert system with the other components of the ECS. This way, instead of being just an assistant, the system will be able to perform actions on behalf of users, e.g. run scripts, send email alerts etc. Secondly, we are also aiming to improve the user experience, by revamping the current GUI, or perhaps even replacing it with a web application.

References

[1] ALICE Collaboration 2008 *JINST*
[2] Gaspar C. Franek B. 1997 *CHEP*
[3] Chapeland S. et al. 2013 *Proceedings to be published in CHEP*