Modelling the Dynamic Behaviour of the ATLAS DAQ prototype - 1 Run Control using Statecharts

P. Croll¹, P. Y. Duval¹, R. Jones¹, S. Kolos², R. F. Sari² and S. Wheeler²
¹CERN, Geneva, Switzerland
²University of Sheffield, Sheffield, England
³Marseille
⁴St. Petersburg (on leave to CERN)

Abstract
Many applications within the ATLAS DAQ prototype-1 have complicated dynamic behaviour which can be successfully modelled in terms of states and transitions between them. Previously, state diagrams, implemented as finite-state machines, have been used. Although effective, they become unwieldy as system size increases. Harel statecharts address this problem by implementing additional features such as hierarchy and concurrency.

The CHSM object-oriented language system is freeware which implements Harel statecharts as concurrent, hierarchical, finite-state machines (CHSMs). An evaluation of the language system by the ATLAS DAQ group has shown it to be suitable for describing the dynamic behaviour of typical DAQ applications and it is currently being used to model the dynamic behaviour of the prototype-1 run control system. The design is specified by means of a CHSM description file and C++ code is obtained by running the CHSM compiler on the file. In parallel with the modelling work, a code generator has been developed which translates statecharts, drawn using the S/P CASE tool, into the CHSM language. Using both the CHSM generator and the CHSM compiler, C++ code describing the dynamic behaviour of the run control system has been successfully generated directly from S/P statecharts. The design has been verified using the simulation features of the Starmate CASE tool.

INTRODUCTION

ATLAS is a particle physics experiment under construction for the Large Hadron Collider (LHC) at CERN. The DAQ system foreseen for this experiment will have to cope with unprecedented data rates (>10 GB/second) and volumes (need more numbers here). In order to meet the challenge, the ATLAS DAQ/EF (Data Acquisition / Event Filter) prototype-1 project aims to produce a fully functional prototype suitable for evaluating candidate architectures and technologies for the final DAQ/EF system of the ATLAS experiment. The prototype consists of a complete "vertical slice" of the ATLAS DAQ/EF architecture. It includes all the hardware and software elements of the data flow chain and the control and monitoring required in an on-line system.

Within the prototype project, the back-end DAQ [ref] encompasses the software needed to configure, control and monitor the DAQ, and excludes the management, processing and transportation of physics data.

The run-control system is one of the software components of the back-end DAQ. It is responsible for controlling the data-taking activities of acquisition components throughout the DAQ system. It can send commands to DAQ components, query or receive status information from them and provides operators with a means to act upon the DAQ system.

The ATLAS collaboration has defined requirements for a software process and a supporting environment to develop and modify all ATLAS software [URD SDE ref]. CASE (Computer Aided Software Engineering) tools using suitable object-oriented modelling methods have been identified as an appropriate technology for supporting many aspects of this process.

This paper begins with a summary of the process presented in more detail elsewhere [Ash CHEP'95 ref], by which suitable methods and CASE tools for the design of the run-control system and other DAQ software components were chosen. An overview of the chosen CASE tools is given. This is followed by a more detailed of the method by which we have used these tools to achieve and subsequently verify the high-level design of the ATLAS DAQ prototype run-control system.

CHOICE OF METHOD AND CASE TOOLS

The ATLAS experiment will demand software production on a scale far beyond that previously addressed in HEP. Due to its size, complexity and forecasted lifetime, it is crucial that software is produced and maintained to the highest possible engineering standards. A considerable shift from the methods traditionally used to produce software in HEP is required. Fortunately, advances in computer science and technology have provided software developers with powerful new tools to face this challenge. Object-oriented (OO) design methods used in conjunction with CASE tools allow the production of robust, re-usable software with sufficient flexibility to meet the changing requirements of an HEP experiment. This section summarises the process by which a suitable OO method and supporting CASE tools
were chosen to design the run-control and other software systems in the DAQ prototype project.

Many applications within the ATLAS DAQ system, including the run-control system, exhibit complicated dynamic behaviour. The system can be successfully modelled in terms of states and transitions between states. In the past, several different methods have been investigated to model such complex dynamic behaviour. These methods are Petri nets, Z-Specification and Finite-State Machines.

Petri nets offer good concurrent modelling facilities coupled with a formal equivalence to underpin its graphical representation. The basic place-transition Petri net suffers from requiring low-level detail to capture a complete system model. Once any non-trivial system has been developed, analysis of the net can suffer from a state explosion problem, i.e. the size of the search space rises exponentially relative to the number of system states. To handle this, Petri net based CASE tools such as Design/CPN have been developed which avoid both substitution transitions (hierarchies) and colours (data typing). Experience to date [Croll 95] has shown that these tools need both a considerable degree of skill from a software engineer together with constraints on the design to make any analysis tractable. Additionally, Petri nets tools lack dependable commercial support and were thus rejected.

Z is a formal method that permits unambiguous mathematical specification. Like many formal methods, Z does not have any direct support for concurrency or simulation and can thus only provide support for part of the software development life-cycle. State diagrams, whereby a dynamic system is modelled in terms of states and transitions caused by events between states, is a very effective representation. However, they become unsatisfactory as system size increases due to a lack of abstraction. Harel statecharts [ref] address this problem by implementing additional features such as hierarchy or state decomposition and concurrency. Statechart has a better method for describing abstraction, default, history, and scale. We therefore decided to use the OMT object-oriented modelling method [ref] which uses Harel statecharts for the dynamic model. In addition to providing a suitable way of modelling dynamic behaviour, OMT allows the static structure to be represented via an object model. By using CASE tools supporting such modelling methods to draw diagrams of the system under study, software developers are able to concentrate their efforts on design aspects rather than implementation.

When choosing a CASE tool for the design of the run-control software, the following points also had to be taken into consideration:

- It should be possible to generate code automatically from the design diagrams to ensure the maintainability of the software over several versions.
- The run-control system has to run in a heterogeneous environment. The software developed with the tool will have to compile and run correctly on different operating systems, presently Solaris, LynxOS, HPUX and WindowsNT. Hence code developed with the tool should be portable and not depend on a run-time library for which source code is not available.

- It must be possible to customise the code generation so that different languages can be produced, for all elements of the model (i.e. not just from the object model) and in order that differences in compilers on different machines can be taken into account.
- It must be possible to integrate generated code with third-party software.
- It should be lightweight, not requiring excessive resources to run.
- It should be easy to learn and to use.

After evaluation of a number of different CASE tools [ref CHEP95], we decided that the Si/POMT (Software through Pictures) CASE tool [ref] was the best overall product for our purposes.

Although code generators can be written and integrated with SiP, allowing C++ code to be generated directly from the dynamic model, an intermediate step, using the CHSM object-oriented language system [ref] has been employed which greatly simplifies the effort. The CHSM language system is summarised in the following section.

**OVERVIEW OF THE CASE TOOLS**

SiP and its code generation capabilities are described in detail elsewhere in this proceeding [Ashraf ref].

**The CHSM Language System**

CHSM is a theoretically-rigorous, object-oriented language system, built on C++, which implements Harel statecharts as Concurrent, Hierarchical, Finite-State machines. The language system supports the following statechart concepts:

- **Hierarchy**: child states can be "nested" in parent states, allowing the parent state to be treated as a "black-box".
- **Clusters**: logical-exclusive-or state groups eliminate the need for replicated transitions.
- **Sets**: logical-and state groups eliminate the exponential increase in the number of states when new states are added.
- **Concurrency**: sets allow transitions caused by the same event to occur simultaneously in different parts of the statechart.
- **History**: On entering a cluster, the statechart enters the child-state that the cluster was in, the last time it was in that cluster.
- **Guard conditions**: transitions are only made if a pre-defined condition is true.
• Actions: actions can be executed when transitions are made and on entering or exiting a state.
• Broadcasting: events can be broadcast when a transition is made.
• Implicit broadcasting: events are broadcast every time states are entered or exited possibly triggering transitions in other parts of the statechart.

The above features can be described in a CHSM description file. This is an ordinary text file consisting of three sections. The first section contains declarations required by any C++ code embedded in the following sections. The second is the CHSM description itself and the third contains optional user code. Guard conditions, transition and state enter and exit actions are specified in the middle section of the description file with the necessary C++ code.

The description file is converted to C++ by the CHSM compiler. The resulting code is compiled and linked with the CHSM run-time library, the source for which is freely available. In the implementation of the run-time library, states, clusters, sets and the CHSM itself are variables of the C++ classes State, Cluster, Set and CHSM respectively. These classes have predefined data-members and member-functions. The user can derive classes from these classes using C++ inheritance in the declaration section of the CHSM description file in order to add data-members and member-functions. Furthermore, existing functions such as state enter and exit functions can be overloaded in order to augment their behaviour. This is a powerful feature allowing C++ classes used in the DAQ system to inherit their dynamic behaviour from CHSM.

There are several independent groups working on the DAQ subsystems. They will need to customise the run controller responsible for their part of the DAQ to perform the operations specific for their particular component. We cannot define in advance the actions which will need to be performed. A generic statechart is defined using SIT and CHSM which the developers use as a "template" framework into which they can add their own specific operations.

Evaluations in the ATLAS DAQ group have proved CHSM to be a very robust and flexible tool. It is a hybrid language where C++ has been extended with additional constructs and, as such, it has proved to be quick and easy to learn. Furthermore, since it is based on C++ , the resultant code is easily incorporated with other programs and modules of the DAQ system. A CHSM code generator has been integrated with the SIT tool to allow CHSM description files to be generated automatically from statecharts in the tool. The code generated by the CHSM compiler, as well as the source for the run-time library, compiles and runs correctly on all the platforms foreseen in the DAQ system. The CHSM code has been successfully integrated with other commercial tools to be used in the DAQ system (e.g. Corba/IDL, X-Windows System, etc.)

ARCHITECTURE OF THE RUN CONTROL

Due to the size and complexity of the ATLAS experiment, the run-control system cannot be implemented as a single program as has been done in previous experiments. It is foreseen that the system will consist of many programs running on a network of computers. Such a distributed system reflects the structure of the DAQ system and will be implemented as a hierarchy of entities called controllers, each with responsibility for a well-defined component of the DAQ system. The controller's state is the simplified external view of the current working condition of the component under its responsibility.

Each controller can receive commands from the outside world. Commands cause a controller to execute actions which potentially change the visible state of the component. A controller can also react to local events occurring in the component under its responsibility. Typically its reaction will be to execute some actions and potentially change its visible state.

The controllers are organised into a hierarchical tree structure which reflects the general organisation of the DAQ system itself. The hierarchy is defined in a configuration database which is described in detail elsewhere in these proceedings [13, ref.]. Each controller in the tree can have one parent (or superior) controller and any number of child (subordinate) controllers. At the top of the tree is a single controller which represents the overall state of the entire system.

The controllers in the hierarchical tree transmit messages between each other over a local-area network (using the OMG Corba standard implemented by IDL/POA) in order to exchange commands and status information. In general, commands starting from the human operator are sent to the overall controller which forwards them to the sub-system controllers who in turn forward them to component controllers and so on. In this respect commands flow up from the root of the tree towards the leaves. Replies indicating the successful completion or otherwise of commands and status information are sent back down the tree so that the human operator is made aware of any change in the state of the system or of any errors which have occurred. Any node in the control tree can perform actions on the commands or result of commands it receives.

HIGHER-LEVEL DESIGN OF THE RUN-CONTROL SYSTEM

The high-level design of the dynamic behaviour of a single run controller was accomplished using the SIT CASE tool. Two statecharts were drawn reflecting different aspects of a run controller's behaviour. It is envisaged that every run controller in the hierarchy will be modelled by the same two statecharts. The first statechart models the sequence of events necessary to take the controlled apparatus from an idle state where no data are being taken, to an active, running state where data are being collected from the apparatus, and back.
again. This statechart is known as the "generic DAQ controller CHSM". This CHSM, extracted from SP, is shown in Figure 1. State changes are initiated by commands corresponding to events on this statechart. These are initiated by a human operator and then propagated through the run control system. It is foreseen that developers of the various controllers will be able to customise the behaviour of their particular controller by adding code to implement the required actions within the generalised template provided by this CHSM.

Referring to Figure 1, the central Alive super-state is composed of two concurrent states: DAQActivity, which reflects the current status of data-taking activities and DAQFault, which reflects whether or not an error has occurred in the component being controlled. The human interface issues commands which correspond to events in the DAQActivity state. When transitions, caused by these events, occur, actions are performed which carry out the necessary operations in order to take the controlled component from one state to the next.

If the action fails for some reason (i.e. the component being controlled does not respond correctly to the action) it can be signalled to the parent controller by issuing the DAQError event which takes the concurrent DAQFault state to Fail. Recovery mechanisms have been envisaged for three different levels of error. Firstly, if an error occurs when making a transition between two states, a mechanism is foreseen to take the CHSM back to the last error-free state undoing any actions which were made during the transition which caused the error. Secondly, if the error is more serious and cannot be cleared by the above mechanism, the whole CHSM can be reset, during which all allocated resources are reset and freed and the CHSM is put back to its initial state. Finally, if a fatal, non-recoverable error occurs somewhere in the overall run control system, the whole system shutdown as cleanly as possible. Since system integrity cannot be guaranteed in such a situation each individual controller cannot rely on any communication or external interaction during the shutdown.

In addition to the generic CHSM, another "Manager CHSM" models the interaction between the different controllers in the run-control hierarchy. The Manager CHSM, after being ported to Statemate, is shown in figure 2. For communication between controllers, we use dedicated ILU [Sergei ed] for sending commands up the tree and the Information Service (another component of the back-end DAQ system) for receiving replies down the tree and for making DAQActivity states visible to outside parties.

The main purpose of the Manager CHSM is to handle the initial configuration of a controller when booting and to marshal the generic CHSM through transitions when DAQ control commands are received from the operator (or a parent controller). This includes forwarding the command to any children a controller may have (either in synchronous or asynchronous mode depending on the nature of the action being carried out). During the initialisation and transition phases, the CHSM is locked so that no new commands can be received.

A concurrent state called RCFault is used to reflect the status of the run control hierarchy itself. If this controller has a problem with any of its children (e.g. a child is dead, not responding, etc.) the RCFault state is set to Fail.

The Membership concurrent state indicates whether the controller is part of the run control hierarchy of controllers. When in the controller will be controlled by its parent controller and ignored when it is Out. This feature allows a controller with a specific problem needing attention to be isolated without affecting the rest of the system.

Testing the High-Level Design

Although the approach we have followed has allowed us to create a high-level design of the run control covering all aspects of the OMT model there was no simple way in which the dynamic model could be simulated in order to test its validity before implementation could begin. To overcome this drawback two different methods of testing have been investigated. The first involved porting the statecharts to the Statemate CASE tool and then using Statemate's powerful simulation facilities to animate the statecharts. We did not use Statemate as the standard design tool because although it supports the OMT dynamic model it uses its own method for the static model. The second test approach involved generating a prototype controller via the CHSM code generator integrated with SP and then using the extensive debugging facilities provided with the CHSM run-time library to check the behaviour.

Overview of the Statemate CASE Tool

Statemate Magnum, a CASE tool from 1-Logic, a company founded by David Harel who introduced the statechart notation, has been used as a simulation tool in testing the high level design in the Atlas project. Statemate is a set of tools with a heavy graphical orientation, intended for the specification, analysis, design and documentation of large and complex reactive systems. Statemate provides an executable semantic of Statechart, can be used to drive a simulation and automatically generate executable code. The statechart model is concerned with the behaviour of objects and their relationships over time. Statemate Magnum is a formal language, where the functional description is the Activity chart, the behavioural description is the Statechart (describes where and how the system performs its function) and the structural description is the Modulechart. The latest information about this tool can be read in [Harel97a].

Statemate aims to provide the necessary features expected of an industrial strength CASE tool. These include powerful graphical editors for statecharts and user-defined control panel, plus simulation and dynamic test facilities. In addition Statemate has an automatic code generator which
can produce C and ADA code directly from the graphical model. Analysis of statechart models is possible to include the behavioural analysis of reachability, deadlock and usage of transitions such as fairness and nondeterminism. However, no formal proof are given.

Statemate Magnum's graphical language and tools have been used to demonstrate product functionality through basic prototyping. Similar to SP-CHSM language system, Statemate has a general syntax of an expression which labels a transition in a statechart, i.e. \textit{event(conditions);action}.

Some minor differences in syntax between the CHSM and Statemate statecharts have been found during the prototype implementation with Statemate.

In Statemate the behavioural view is implemented in the graphical language of statecharts. Among the Activity Chart, Module Chart and Statechart available in Statemate. Only the statechart view was used since this matched the existing design available with SP and CHSM. The statechart behavioural view, captures the when. It describes the system's behaviour over time, including the dynamics of activities, their control and timing behaviour, the states and modes of the system, and the conditions and events that cause modes to change and other occurrences to take place.

The behavioural model provides answers to questions about causality, concurrence and synchronisation, which are paramount features for the Run Control System.

Statecharts incorporate a broadcast communication mechanism, time-out and delay operators for specifying synchronisation and timing information. Each element in the Statechart has an entry in the Data Dictionary, which can be used to put additional specific information. Statemate allows splitting large charts into separate hierarchical ones. This feature has been used for the two communicating statecharts namely the Generic Run Controller and RC FSM Manager, see figure 1 and figure 2. Statemate allows shared information in a model component called Global Definition Set. This similar to scoping in programming languages.

The scope of the statechart can easily be shown due to the presence of the control symbol (\texttt{C}) or instantiation symbol (\texttt{I}) that inside a state there is another decomposed component statechart. This facility enables the designer to improve readability by hiding part of the structure in another diagram and to promote reuse of statecharts. For example, the excCond state in this RC FSM Manager statechart (see figure 2) referred to the generic RC statechart (see figure 1).

The Statemate data dictionary editor have been used to configure the binding of the statecharts objects, and to create an array of objects to show which instantiations need to be controlled. The data dictionary editor can be used to configure the data structure, data type and usage of an object, and also whether it is an IN or OUT parameter. The Statemate querying tool has been found efficient, as it can generate lists to help in finding an object in a large system and in finding where it is referenced.

**Simulation with Statemate**

Changes between state in a statechart can be simulated by Statemate Simulator, where a currently running transition or state will be indicated by different colour from the other states. Therefore, the system’s behaviour can easily be observed because it is visible. Before a simulation can be run, the scope of the simulation profile will check, the correctness of the statecharts to show detected inconsistencies, while the completeness check detects redundancy and incompleteness of the model. Hence, the violation in syntax and semantic on both in a single chart and amongst the charts can be tested.

Statemate offers the use of interactive and batch mode of simulation. The batch mode simulation tools can be the entry of large amounts of data. A scenario-based execution can be better described with these features. Interactive mode means that simulation control in the form of GO command, and element value change is done manually, either by typing commands or by selecting commands from the Simulation menu. Batch mode simulation means that Simulation is controlled by the main section of an Simulation Control program (SCP). This is a recorded sequence of simulation on a particular process environment.

The simulation of the run statecharts of the Run Control system is able to show the interaction between the statecharts. However, the interaction between the hierarchy is difficult to observe because of the limited screen space for both the statecharts and their control panels to indicate the changes of state. Therefore, the communication between the layers in the hierarchy of control in the Atlas Run Control is difficult to observe.

**Lesson learnt from Using Statemate**

The approach which enables binding different object names and using the instantiation features of Statemate for simulating different statecharts has been implemented. However, the implementation of the system in a distributed environment with a real implementation has to be validated. Therefore, not all the features of the hierarchy of control have been simulated and analysed. More experiments are needed to test the use of the hierarchy of statecharts of the Run Control system. In addition, not all possible normal and abnormal condition that might occur during program execution can be tested.

Some benefit of Statemate Magnum in enhancing the overall software engineering cycle have been learnt from this project, such as, it simplifies modelling of complex system, eliminates ambiguities common in textual specification, it validates system behaviour, and it detects and eliminates specification errors before implementation.

The most helpful feature of Statemate in this study is its capability to simplify understanding of operation with clear animation of graphical models. CASE tool such as Statemate can be used to show what would happen if multiple features took place, which is sometimes hard to conceptualise.
Testing with CHSM

The CHSM code generator integrated with SfP was used to create C++ code corresponding to the high-level design. The code was compiled and linked with the CHSM run-time library in order to get a working test program. The CHSM run-time library contains two debug features. The first is called dump state. This prints to the screen the current state of the CHSM. It can be called at anytime to ascertain which states in the CHSM are currently active.

The second debug feature prints out a configurable amount of information execution of the program including a report of state entrances and exits, of event queueing and dequeuing, and of the actions performed during the transition. Using these debug features it was found, for instance, that the concurrent state in the Manager CHSM, responsible for packing the CHSM when handling a transition, was not synchronised properly. Unfortunately, in order to carry out more extensive tests it has been necessary to add a significant amount of hand-written code to implement the controller hierarchy.

CONCLUSIONS

We have found Harel statecharts to be an excellent method for modelling the dynamic behaviour of the run control and we have used a combination of CASE tools, that support statecharts, to create and test the high-level run control design. SfP provided the necessary formalism and its code-generation facilities combined with the CHSM language system gave sufficient flexibility to allow third-party software to be successfully integrated, to enable the generated software to run in a heterogeneous environment and to allow each controller in the system to be customised. The extensive debugging facilities of the CHSM language system combined with the simulation facilities of StateMate allowed us to test the high-level design before embarking on the detailed implementation. Now that the high-level design is complete a prototype run controller has been created and integrated with third-party software to implement communication between controllers and is currently undergoing thorough testing using the CHSM language-system debug facilities.

For future work we are investigating the possibility of automatically porting the design from SfP to StateMate. For the current project, the porting was done by hand. One problem we have encountered with StateMate is that it cannot be used to simulate the distributed run controller hierarchy. Although we have been able to verify the model for a single run controller we have not been able to create a simulation whereby multiple copies of the run controller statecharts representing different processes running on different machines, interact in a hierarchy. We hope that Rhapsody (the successor to StateMate) may overcome some of these difficulties. Rhapsody is a new CASE tool which was launched in May 1997 by I-Logix as an integrated set of diagrammatic languages for object modelling, built around statecharts [Harel, July 1997]. This tool addresses the issue of developing an executable object modeling with statecharts and to produce an object orientated C code such as C++. We plan to investigate the use of Rhapsody with the problem.

ACKNOWLEDGEMENTS

We would like to thank Paul J. Lucas for answering many questions about the CHSM language-system and his considerable help in porting the system to Windows NT.

REFERENCES


Figure 2. The BC_FSM_Manager Statechart with Statome