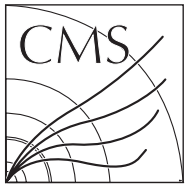
Available on CMS information server

CMS CR -2010/301



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

16 December 2010

An Analysis of the Control Hierarchy Modeling of the CMS Detector Control System

Yi Ling Hwong for the CMS Collaboration

Abstract

The supervisory level of the Detector Control System (DCS) of the CMS experiment is implemented using Finite State Machines (FSM), which model the behaviors and control the operations of all the sub-detectors and support services. The FSM tree of the whole CMS experiment consists of more than 30.000 nodes. An analysis of a system of such size is a complex task but is a crucial step towards the improvement of the overall performance of the FSM system. This paper presents the analysis of the CMS FSM system using the micro Common Representation Language 2 (mcrl2) methodology. Individual mCRL2 models are obtained for the FSM systems of the CMS sub-detectors using the ASF+SDF automated translation tool. Different mCRL2 operations are applied to the mCRL2 models. A mCRL2 simulation tool is used to closer examine the system. Visualization of a system based on the exploration of its state space is enabled with a mCRL2 tool. Requirements such as command and state propagation are expressed using modal mu-calculus and checked using a model checking algorithm. For checking local requirements such as endless loop freedom, the Bounded Model Checking technique is applied. This paper discusses these analysis techniques and presents the results of their application on the CMS FSM system.

Presented at *CHEP2010: International Conference on Computing in High Energy and Nuclear Physics 2010*

An Analysis of the Control Hierarchy Modelling of the CMS Detector Control System

Yi Ling Hwong[2], Tim Willemse[8], Vincent Kusters[8], Gerry Bauer[7], Barbara Beccati[2], Ulf Behrens[1], Kurt Biery[6], Olivier Bouffet[2], James Branson[5], Sebastian Bukowiec[2], Eric Cano[2], Harry Cheung[6], Marek Ciganek[2], Sergio Cittolin[a,2], Jose Antonio Coarasa[2], Christian Deldicque[2], Aymeric Dupont[2], Samim Erhan[4], Dominique Gigi[2], Frank Glege[2], Robert Gomez-Reino[2], Andre Holzner[5], Derek Hatton[1], Lorenzo Masetti[2], Frans Meijers[2], Emilio Meschi[2], Remigius K. Mommsen[6], Roland Moser[2], Vivian O'Dell[6], Luciano Orsini[2], Christoph Paus[7], Andrea Petrucci[2], Marco Pieri[5], Attila Racz[2], Olivier Raginel[7], Hannes Sakulin[2], Matteo Sani[5], Philipp Schieferdecker[a,2], Christoph Schwick[2], Dennis Shpakov[6], Michal Simon[2], Konstanty Sumorok[7]

[1]DESY, Hamburg, Germany, [2]CERN, Geneva, Switzerland, [3]Eidgenössische Technische Hochschule, Zurich, Switzerland, [4]University of California, Los Angeles, Los Angeles, California, USA, [5]University of California, San Diego, San Diego, California, USA, [6]FNAL, Chicago, Illinois, USA, [7]Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, [8]Technical University of Eindhoven, [a]Now at Universitaet Karlsruhe

Email: Yi Ling Hwong - yi.ling.hwong@cern.ch

Abstract. The supervisory level of the Detector Control System (DCS) of the CMS experiment is implemented using Finite State Machines (FSM), which model the behaviours and control the operations of all the sub-detectors and support services. The FSM tree of the whole CMS experiment consists of more than 30.000 nodes. An analysis of a system of such size is a complex task but is a crucial step towards the improvement of the overall performance of the DCS system. This paper presents an analysis of the CMS FSM system using the micro Common Representation Language 2 (mCRL2) methodology. Individual mCRL2 models are obtained for the FSM systems of the CMS sub-detectors using ASF+SDF. Different mCRL2 operations are applied to the mCRL2 models. A mCRL2 simulation tool is used to closer examine the system. Visualization of a system based on the exploration of its state space is enabled with a mCRL2 tool. Requirements such as command and state propagation are expressed using modal mu-calculus and checked using a model checking algorithm. For checking local requirements such as endless loop freedom, the Bounded Model Checking technique is applied. This paper discusses these analysis techniques and presents the results of their application on the CMS FSM system.

1. Introduction

The CMS experiment is a general purpose detector designed to study a wide range of particles and phenomena produced in the high-energy collisions in the LHC. The control, configuration, readout and monitoring of hardware devices as well as the monitoring of external systems, such as the electrical system, cooling system, etc, is carried out by the Detector Control System (DCS). The modeling of the control system is implemented as a hierarchy of Finite State Machines (FSM) and is developed using the SMI++ toolkit [1]. FSMs are described using the State Manager Language (SML) and are organized in a tree structure where each node has one parent and zero or more children, except for the top node, which has no parent. Commands are sent from a parent node down to its children and the states of the children are propagated up the tree (Figure 1). The FSM tree of the whole CMS experiment consists of more than 30.000 nodes, which makes the design and implementation of a homogenous and consistent system a complex task. A single badly designed FSM may be sufficient to lead to endless loops, potentially hampering the performance of the experiment.

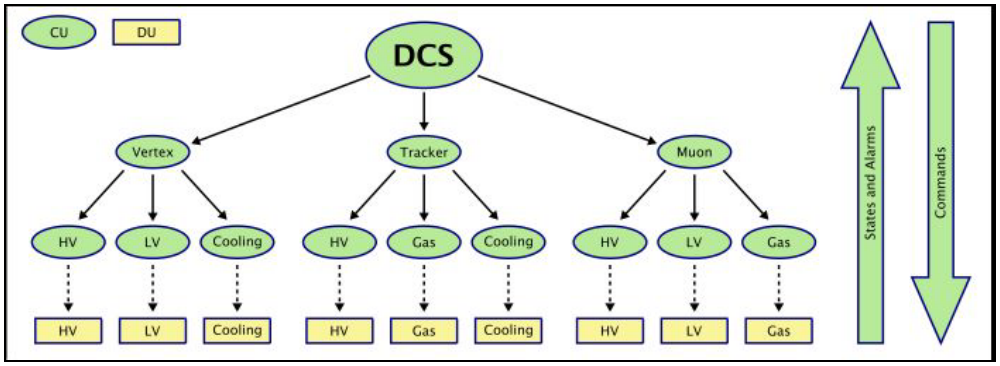


Figure 1. A simple control system modeled using Finite State Machines

In view of the complexity of the FSM system, it is almost impossible to track down problems when unexpected behaviors manifest themselves. In this project we developed automated verification tooling for our analysis purposes. The project is divided into two main parts. First, we formalised the SML by mapping its language construct onto constructs in the process algebraic language mCRL2 [2]. Second, we identified properties that can be verified for FSMs in isolation and developed dedicated verification tooling based on Bounded Model Checking [3].

Using the ASF+SDF meta-environment [4], we developed a prototype translation implementing our mapping of SML to mCRL2. This allowed us to quickly assess the correctness of the translation through simulation and visualization of FSMs in isolation. Our dedicated verification tools allow the FSM developers to quickly perform behavioral sanity check on their code and use the feedback to further improve on their design.

2. The State Manager Language (SML)

The Finite State Machines (FSMs) are described using the State Manager Language which is provided by the SMI++ toolkit. Listing 1 shows a snippet of the definition for a *class* in SML.

A *class* is the declaration of a FSM object and consists of one or more *state clauses*. Each state clause consists of zero or more *when clauses* and *action clauses*. The *when clauses* describe how the object will behave in a certain state, *i.e.* the rules by which the object will obey for transition between states or to execute a command. A *when clause* has two parts: a *guard* which is a Boolean constraint on the states of the children of the object and a *referrer* which describes what should happen if the guard evaluates to true. An *action clause* consists of a *name* and a list of *statements*. When an object in a state *S* receives a command from its parent, it looks inside the *state clause* of state S for an *action clause* with the same name as the command and if such an *action clause* exists, it executes its statement list. Commands received which are not declared in the *action clause* are ignored.

```
class: $FWPART_$TOP$RPC_Wheel_CLASS
    state: OFF
     when ( $ANY$FwCHILDREN in_state ERROR )  move_to ERROR
     when ( $ANY$FwCHILDREN in_state RAMPING )  move_to RAMPING
     when ( $ALL$FwCHILDREN in_state STANDBY )  move_to STANDBY
     when ( $ALL$FwCHILDREN in_state ON )  move_to ON
     when ( ( $ALL$FwCHILDREN not_in_state OFF ) and
            ( $ANY$FwCHILDREN in_state STANDBY ) )  move_to STANDBY

        action: STANDBY
            do STANDBY $ALL$FwCHILDREN
        action: OFF
            do OFF $ALL$FwCHILDREN
        action: ON
            do ON $ALL$FwCHILDREN
```

Listing 1. Part of the definition of the *Wheel* class in SML

2.1. From SML to mCRL2

We formalise the semantics of programs written in SML using the process algebra mCRL2 by providing a mathematical model for the language constructs of SML. Consequently the SML codes can be analysed mathematically. mCRL2 is a specification language that can be used to specify and analyse the behaviour of large distributed systems. The language is supported by a toolset enabling simulation, visualisation and verification of software requirements.

We utilised the ASF+SDF meta-environment to develop an automatic translator which will translate SML into mCRL2. This allows us to model a FSM model in the mCRL2 language, upon which the verification tool from the mCRL2 toolset is applied (see Section 3). The *Syntax Definition Formalism* (SDF) was used to describe the syntax of both SML and mCRL2, whereas the *Algebraic Specification Formalism* (ASF) was used to express the term rewrite rules for the actual translation. An example of the translation is shown in Translation 1.

SML	**mCRL2**

State: OFF
 	
when G1 move_to S1
…
when Gn move_to Sn
	inState_OFF(s) && isWhenPhase(phase) -> (
	  translation_of_G1 ->
	  move_state(self,S1).
	  Wheel(self,parent,S1,chs,phase,aArgs) <>
	  …
	  translation_of_Gn ->
	  move_state(self,Sn).
	  Wheel(self,parent,Sn,chs,phase,aArgs) <>
	  send_state(self,parent,s).
	  move_phase(self,ActionPhase).
	  Wheel(self,parent,s,chs,ActionPhase,
	  reset(aArgs)))

Translation 1. A simplified translation of the *when clauses* of a state OFF for the *Wheel* class

Due to the intricacies of the language, the formalisation of SML was far from trivial. We used the mCRL2 simulation and visualization tools to ensure a translation that reflects the real system accurately. This is an important part of the project as subsequent model checking using the mCRL2

verification tool is performed on this mCRL2 model derived from the translation. It is crucial that the translated model is as close to the actual behaviour as possible because otherwise, the outcome of the analysis of the models which is conducted using mCRL2 (or BMC, which will be discussed in section 4) may not reflect what goes on in the real system.

3. General Tooling for Verification

The analysis of a system often involves showing that the modelled system exhibits certain desired properties, or does not exhibit an undesired one. A powerful verification method which is supported in the mCRL2 toolset is model checking. A formula expressing a desired property that the system should not violate (or satisfy) is needed for model checking. Such formulas are expressed in regular modal mu-calculus [5]. The formulation of requirements is an involved matter and often requires several attempts before the desired property is accurately expressed. To this end we have formulated some basic requirements and check them on the *Wheel* subsystem, see Table 1.

Table 1. Basic requirements for the *Wheel* subsystem; `i:Id` denotes an identifier of an FSM; `i_c:Id` denotes a child of FSM `i`; `c:Command` denotes a command; `c2s(c)` denotes the state with the homonymous command name, *e.g.*, `c2s(ON) = ON`

1. Absence of deadlock:
```
nu X. [true]X && <true>true
```
2. Absence of endless loop:
```
nu X. [true]X &&
      [exists s:State. move_state(i,s)](nu Y
      [(!move_phase(i,ActionPhase))]Y &&
      [exists s:State. move_state(i,s)] false)
```
3. Responsiveness:
```
nu X. [true]X && [comm_command(i,i_c,c)](mu Y. <true>true &&
      [!comm_state(i_c,i,c2s(c))||
        exists c':Command. comm._command(i,i_c,c'))]Y)
```
4. Progress:
```
nu X. [true]X && mu Y. <exists s:State. move_state(i,s)>true ||
      [true]Y
```

The first two requirements are fairly straightforward while the third and forth requirements are more elaborate. The 'Responsiveness' requirement expresses the inevitability of a state change by a child once such a state change has been commissioned while the 'Progress' requirement asserts that state changes should always be attainable, *i.e.*, an FSM will eventually change its state. All these requirements hold for the *Wheel* subsystem.

The expressiveness of the modal mu-calculus entails the versatility of the mCRL2 verification tool to check for any requirements. However, the enormous state-space of the FSM system renders this a time consuming process.

4. Dedicated Tooling for Verification

In an attempt to improve on the time performance of a general verification tooling, we explored the possibilities of using *Bounded Model Checking* (BMC) to analyse a FSM in isolation. The basic idea of BMC is to check for a counterexample in bounded runs. If no bugs are found using the current bound, then the bound is increased until either a bug is found, the problem becomes intractable, or

some pre-determined upper bound is reached upon which the verification is complete. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved using a SAT solving tool [6]. We have applied BMC techniques for the detection of endless *move_to* loops and the detection of unreachable states and trap states. An example of an endless *move_to* loop can be seen in Listing 2, showing an excerpt of the `ECALfw_CoolingDee` SML code which our tool found to contain issues. If an instance of `ECALfw_CoolingDee` has one child in state `ERROR` and one child in state `NO_CONNECTION`, it will loop indefinitely between these two states.

```
state: ERROR
  when ( $ANY$FwCHILDREN in_state NO_CONNECTION ) move_to NO_CONNECTION
  when ( $ALL$FwCHILDREN in_state OK ) move_to OK


state: NO_CONNECTION
  when ( $ALL$FwCHILDREN in_state OK ) move_to OK
  when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
```

Listing 2. An excerpt from the `ECALfw_CoolingDee` SML code that exhibits an endless loop

We formulated the problem of detecting a loop into a SAT problem with the predicate *in_state* defined as follows: *in_state(s,p,i)* holds if and only if the process with identifier *p* is in state *s* after *i* steps. We assign the identifier zero to the FSM under consideration. The resulting formula has three components: the *state constraints*, the *transition relation* and the *loop condition*. The state constraints ensure the FSM is always in exactly one state and that the states of the children do not change during the execution of the *when phase*. The transition relation is the translation of the *when clauses*, denoting the *move_to* steps that a FSM is allowed to take. The loop condition states that if *in_state(s,0,0)* holds, then *in_state(s,0,i)* must hold for $i > 1$, indicating that the parent returns to the state in which it started.

For the detection of unreachable states and trap states, we adopted a graphical approach. For this, we determine whether there is a configuration of children such that a FSM *F* can execute a *move_to* action from a state *s* to a state *s'*. Doing so for all pairs (*s*,*s'*) of states of *F* yields a graph encoding all possible state changes of *F*. Computing the strongly connected components (SCCs) of the thusly obtained state change graph gives us a clear overview of the reachability of the FSM states. A well designed FSM should contain only one single SCC. A system displaying more than one SCC is often an indication of a faulty design. The state change graph of the `ESfw_Endcap` FSM class is shown in Figure 2. We can clearly see that the state `OFF` can never be reached from any of the other states.

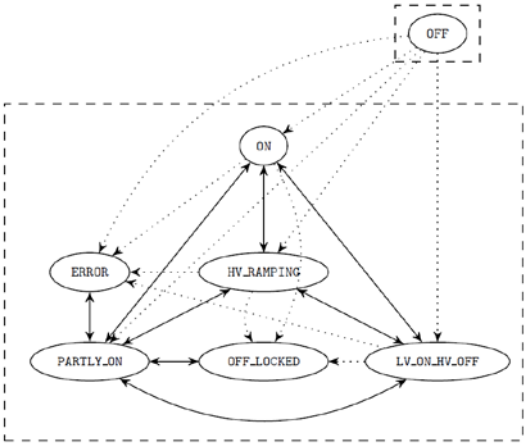


Figure 2. The state change graph of the `ESfw_Endcap` FSM class. The SCCs are placed within a dashed frame

5. Results

We obtained very encouraging results with our dedicated tools for verification. We have analysed more than 40 FSM classes so far and found 6 to contain issues, two of them shown in Listing 2 and Figure 2 respectively. Note that although trivial at first glance, such errors are difficult to be detected due to the size of an average FSM class which contains more than 100 lines of SML code in general. We automated the process of pinpointing the source of an error and this is done in typically less than one second. Our state change graph verification method has been incorporated in the FSM 3D visualization tool which is currently being developed.

6. Conclusion

The Finite State Machine (FSM) system of the CMS experiment is a large and complex system. We studied the State Manager Language (SML) which is used to describe the FSMs and formalised it using the process algebraic language mCRL2. The translation of SML into mCRL2 was implemented using the ASF+SDF meta-environment. We carried out formal verification of a small subsystem using the model checking tool provided by mCRL2. Based on specific needs for certain requirements such as endless loops freedom and the reachability of all states, we also built dedicated tools based on Bounded Model Checking for verification of FSMs in isolation. Results so far are very encouraging and we plan to further explore SAT solving techniques, and combine it with symbolic verification techniques such as the ones currently offered in the mCRL2 and LTSmin toolsets [7].

7. Acknowledgement

This work has been supported in part by a Marie Curie Initial Training Network Fellowship of the European Community's Seventh framework program under contract number(PITN-GA-2008-211801-ACEOLE). We would also like to thank Giel Oerlemans, Dennis Schunselaar and Frank Staals from the Eindhoven University of Technology for their contribution to the automatic translator. We also thank Clara Gaspar for her valuable advice about the SML language.

References

[1] B. Franek and C. Gaspar. SMI++ object-oriented framework for designing and implementing distributed control systems. IEEE Transactions on Nuclear Science, 52(4):891-895, 2005

[2] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. v. Weerdenburg. Analysis of distributed systems with mCRL2. In Process Algebra for Parallel and Distributed Processing, pages 99–128. Chapman Hall, 2009

[3] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. Advances in Computers, 58:118–149, 2003

[4] M. Van den Brand, A. Van Deursen, J. Heering, H.A. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, Proc. of Compiler Construction, volume 2027 of LNCS, pages 365–370. Springer, 2001

[5] J.F. Groote and T.A.C.Willemse. Model-checking processes with data. Science of Computer Programming, 56(3):251–273, 2005

[6] A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking Using Formal Methods in System Design, 19, 7–34, 2001

[7] S.C.C. Blom, B. Lisser, J.C. van de Pol, and M. Weber. A database approach to distributed state-space generation. Journal of Logic and Computation, 2009