Towards the application of a model based design methodology for reliable control systems on HEP experiments

BARROCA, Bruno, et al.

Abstract

The software development process of user interfaces for complex control system can constantly change in requirements. In those systems changes are costly (time consuming) and error prone, since we must guarantee that the resulting system implementation will still be robust and reliable. A way to tackle this problem is to bring a software model based approach for specification and providing at the same time rapid prototyping capabilities (to speed up design) and Simulation/Verification capabilities (to assure quality). We propose a full model-based methodology to guide designers through specification changes. As a validation case study we have chosen a real life problem: the ATLAS Online Software framework, which has the main purpose to provide a stable software development platform for managing both trigger and data-acquisition processes. In this paper, we explain how to describe a particular control system: the ATLAS Trigger-DAQ system and its graphical user interface (GUI), and we describe how we can validate such specifications, by means of automatic verification techniques like simulation and model checking or [...]

Reference


DOI : 10.1109/NSSMIC.2008.4774651
Towards the application of a model based design methodology for reliable control systems on HEP experiments


Abstract—The software development process of user interfaces for complex control systems is the prototype refinement following system designer’s feedback. The unexpected patterns caused by constantly changing human requirements (motivated by their problems), cause frequent changes in the implementation. This changes are costly, and usually lead to erroneous specifications which in limit, could compromise the control system’s security status. The activity of software development brings, of course, some considerable effort to the designers, since they will have to learn about both the existing system’s structure and individual behaviour of each components in order to be able to effectively make any change.

We propose a full model based design methodology to guide designers through specification changes. This guidance can be classified as static and dynamic guidance. Static guidance is achieved by the semi-automated development of design tools (according to a specific domain meta-model), which bounds and assists the specification of a control system. Dynamic guidance is achieved by model-based simulation: this gives the users an evolving view of the possible implications on each specification change. Applied by designers in particular, it gives them an effective way to validate the computational meaning of each design change (e.g debugging facilities, automatic verification and model checking, etc). On the other hand, this model-based simulator can also easily provide a secure-sandbox for the control system operators themselves [10] for training purposes, and to predict the results of their actions on the real system. These guidance capabilities can be assembled in an integrated development environment (IDE), in order to properly aid the designers on each design step.

A clear formal specification of both the structure and behaviour of the system is therefore extremely important not only for documentation purposes (and its relevance to controlling the designer’s learning curve), but also for simulation purposes (and its relevance to establishing quality metrics over the system’s specification). However, this turns out to be difficult to achieve, since control system’s software developers usually work in an heterogeneous environment, meaning that they access through resources and complex algorithms, through platform-dependent framework libraries.

Our goal now is to review this model based approach by means of a comparison with an existing control system’s development framework, in order to an empirical contact with new usable ways to deal with complexity.

In this communication, we will explain how to describe a particular control system: the ATLAS Trigger-DAQ system, by means of the resulting formalisms of the BATICCCS research project, while explaining the inherent relationship of some entities of the specification formalism with platform specific components of the generated GUI prototype. Finally, we will describe how we can validate such specifications, by means of some automatic verification techniques like simulation and model checking or testing.

I. INTRODUCTION

One of the critical phases of developing user interfaces for complex control systems is the prototype refinement following system designer’s feedback. The unexpected patterns caused by constantly changing human requirements (motivated by their problems), cause frequent changes in the implementation. This changes are costly, and usually lead to erroneous specifications which in limit, could compromise the control system’s security status. The activity of software development brings, of course, some considerable effort to the designers, since they will have to learn about both the existing system’s structure and individual behaviour of each components in order to be able to effectively make any change.

We propose a full model based design methodology to guide designers through specification changes. This guidance can be classified as static and dynamic guidance. Static guidance is achieved by the semi-automated development of design tools (according to a specific domain meta-model), which bounds and assists the specification of a control system.

Dynamic guidance is achieved by model-based simulation: this gives the users an evolving view of the possible implications on each specification change. Applied by designers in particular, it gives them an effective way to validate the computational meaning of each design change (e.g debugging facilities, automatic verification and model checking, etc). On the other hand, this model-based simulator can also easily provide a secure-sandbox for the control system operators themselves [10] for training purposes, and to predict the results of their actions on the real system. These guidance capabilities can be assembled in an integrated development environment (IDE), in order to properly aid the designers on each design step.

A clear formal specification of both the structure and behaviour of the system is therefore extremely important not only for documentation purposes (and its relevance to controlling the designer’s learning curve), but also for simulation purposes (and its relevance to establishing quality metrics over the system’s specification). However, this turns out to be difficult to achieve, since control system’s software developers usually work in an heterogeneous environment, meaning that they access through resources and complex algorithms, through platform-dependent framework libraries.

Our goal now is to review this model based approach by means of a comparison with an existing control system’s development framework, in order to an empirical contact with new usable ways to deal with complexity.

In this communication, we will explain how to describe a particular control system: the ATLAS Trigger-DAQ system, by means of the resulting formalisms of the BATICCCS research project, while explaining the inherent relationship of some entities of the specification formalism with platform specific components of the generated GUI prototype. Finally, we will describe how we can validate such specifications, by means of some automatic verification techniques like simulation and model checking or testing.

II. RELATED WORK AND CASE STUDIES

We will now present the two software development frameworks involved in our study.

A. ATLAS Online Software Framework

The ATLAS Online Software Framework (AOS) is responsible for the development and maintenance of the overall ATLAS experiment. It includes database management for the distributed information management, graphical user interfaces for control and configuration, and an online monitoring infrastructure. The software development...
methodology is based, on configuration files, database schemas, and API's which guarantee a certain degree of behavioural conformance between the applications. In an implementation level, the behavioural aspect of components is specified using an imperative paradigm (either in C++ or Java). The software development is guided by an agreement with both the implementation of commonly known AOS Application Programming Interfaces (APIs), and the existing database schemas of the ATLAS Trigger-DAQ control system.

B. Project BATICCCS Framework

The BATICCCS (Building Adaptive Three-Dimensional Interfaces for Critical Complex Control Systems) research project started with the main goal of building a comprehensive model based design methodology in order to enable rapid prototyping of adaptive three-dimensional user interfaces for controlling and diagnosing control systems. The main research topics were: Graphical User Interfaces (GUI engines, both prototyping of adaptive three-dimensional user interfaces for operational and design formalisms), software modelling and controlling and diagnosing control systems. From the research on the topic of providing formal methods in model based design methodology in order to enable rapid prototyping facilities that the designers will have at their service on their development environment, hence providing a true design-guidance mechanism. This characteristic can be easily achieved by the application of meta-modelling tools [6],[5],[9] over the a domain specific modelling language metamodel, in order to automatically produce a usable editor to aid the design of the system model specification. Moreover, using the GMF metamodeling tool, we have generated an

In figure 1, it is shown the top-level of the abstract syntax of the (H)ALL. The message definitions are global concepts since component from different domains might be able to communicate with each other, as happens with task synchronizations.

The structure of all of the above models is defined by a parent-child relationship, where children can be controlled by parent, and where parent states can be updated from child states. To define the components' behaviour, the (H)ALL language uses some well-known computational abstractions. The individual component's behaviour is described by the finite state machine abstraction: representing all possible states of the component or type, and all the possible transitions. The interaction between component's is described by the command/event abstraction: representing all possible command/event communications (this includes both their activation preconditions and deactivation posconditions) between all components in some hierarchical level (e.g a subsystem). The events are broadcasted from higher levels to the lower levels in the hierarchy, and the response events are propagated back from the lower levels. Furthermore, the hierarchical structure of components determines the partial order relation between the handling of these events: e.g an event produced at level 3 is handled first at level 2 which in tum can enable the possibility to be handled by level 1.

In figure 2, it is show a conceptual view of the BATICCCS methodology. We plan to generate models for simulation and execution of both the system plus its GUI. The feedback from the analysis simulation will enable the designers to correct their specification flaws. Moreover, we plan to integrate the verification analysis during the runtime, in order to give the operators a sandbox which gives the possibility to simulate and predict the consequences of their actions [13].
From a source-code analysis on the existing ATLAS IGUI application, we extracted a (H)ALL’s system simulator model, containing the visual model and the system model from a particular ATLAS partition database file. In figure 3, it is shown a system’s structure expressed using the (H)ALL language. Each one of these components share the same behaviour, which is here represented in a representative component: SimuNode, for readability purposes. The behaviour definition of each component is represented by a finite state machine (see figure 4). The states are represented as ellipses and transitions are represented as arrows, and the rectangles represent the message handlers which will trigger each transition. This representation of the system component’s behaviour gives us the possibility to simulate the system’s behaviour without having a connection with the real control system. Furthermore, each system component’s will have a data structure to represent it’s actual name (not represented here).

Fig. 2. The BATICCCS methodology for rapid prototyping of user interfaces for control systems using (H)ALL

editor for the (H)ALL based uniquely on its metamodel.

1) Model for analysis: Having our system and GUI specification well defined syntactically in terms of (H)ALL’s meta-model, enable us to produce an equivalent specification expressed in terms of CO-OPN (an object-oriented petri-net language) [4], where we can automatically generate a simulator to explore the state-space of both the system’s and GUI behaviour. This was done by establishing a general correspondence, between the syntactic entities found in (H)ALL and syntactic entities found in the CO-OPN (e.g. places, transitions, tokens, etc.). This correspondence is currently being implemented by the application of a model transformation language: Atlas Transformation Language (ATL) [2].

2) Model for execution: Having our system and GUI specification expressed in terms of CO-OPN also enable us to generate java code that will act as a controller to communicate with platform specific drivers. The abstract graphical objects internal and external behaviour are matched (via a platform specific driver) with certain implementation artefacts on the graphical engine (e.g. java swing classes). The same communication is applicable between the simulator and the control system’s drivers.

III. APPLICATION OF THE BATICCCS METHODOLOGY

The ATLAS Trigger-DAQ system consists in a set of isolated partitions. In each partition, a certain hierarchical group of trigger and data-aquisition processes can run and communicate independently from the other partitions. The AOS provides an API to structure and guide the development of user interfaces which are suitable to communicate with the overall AOS applications. The Integrated Graphical User Interface (IGUI) is one of the applications built on top of this API. Besides the functionality of monitoring/histogramming and managing the known components of some partition (IS, configuration database, etc.), the IGUI is also able to send commands to the processes in the partition, and monitor their states as well. This is done through the RunControl panel. The RunControl panel is the main component of the IGUI, since it provides a clear and usable control interface over the control system.

From a source-code analysis on the existing ATLAS IGUI application, we extracted a (H)ALL’s system simulator model, containing the visual model and the system model from a particular ATLAS partition database file. In figure 3, it is shown a system’s structure expressed using the (H)ALL language. Each one of these components share the same behaviour, which is here represented in a representative component: SimuNode, for readability purposes. The behaviour definition of each component is represented by a finite state machine (see figure 4). The states are represented as ellipses and transitions are represented as arrows, and the rectangles represent the message handlers which will trigger each transition. This representation of the system component’s behaviour gives us the possibility to simulate the system’s behaviour without having a connection with the real control system. Furthermore, each system component’s will have a data structure to represent it’s actual name (not represented here).

Fig. 3. A partition structure model expressed as (H)ALL system components.
From a preliminary analysis on the AOS API, we have found that it does not impose any separation of concerns in the development of the GUI, i.e. the user interface code can be mixed with domain concepts like process states and status. It is commonly known that the explicit separation of domain concepts from the GUI concepts can make the resulting interface more flexible to adaptation, and can bring us more satisfactory results while performing verification/validation of the entire system+GUI.

We also extracted the (H)ALL’s visual model from the existing IGUI application’s RunControl panel, with same system’s configuration (shown in figures 5 and 6). Since all buttons will share the same interface behaviour, the buttons behaviour was here defined (for readability purposes of this paper) using a representative component called Button (the same abstraction was applied on the Label, and TreeItem representatives). Notice that the behaviour of these graphical components are usually very simple, and having a behaviour model of these, will also enable us to have a simulation of the whole system+GUI.
To connect and synchronize both the system's and visual models, we also needed to extract the task model from the IGU application's RunControl panel. In figure 7, it is described the structure and behaviour of the available tasks on the RunControl panel. The behaviour of the DoSelectTreeNode task is to receive select messages from the GUI's tree panel and send update messages to the command panel to show the available commands for the selected tree node. The DoSelectTreeNode data structure (representing the information about the selected node) will also be propagated to the subtasks DoStateChange and DoStateRefresh. The task DoStateChange is responsible to receive messages from the command panel and translate these commands (according to the selected node) to messages which are able to be interpreted by the system model hierarchy. When some system component changes its state, a message StateChanged is propagated out of the system's hierarchy and is handled by the DoStateChange and DoStateRefresh task. While handling this message, the DoStateChange task will become unblocked, which means that the interface will be able to deal with further commands from the command panel. On the other hand, the DoStateRefresh task, will re-interpret the StateChanged message into messages that will change the properties and states of the affected graphical objects.

This source-code analysis was supported by the construction of a small prototype (MyPanel) which consists in both a GUI and a simulator of the original under study (as seen in figure 8), and is statically conformant with the (H)ALL models presented above.

A. Building Adaptive Interfaces

The resulting visual model is generic and totally independent from the simulator concepts. The MyPanel prototype can read a partition database file and create the simulator nodes with their respective visual objects. However, the (H)ALL language is not able to express the dynamic construction of both system and visual components from a partition database file, since it involves the creation of new model entities. An (H)ALL specification is a model of a static interface: it contains the sufficient information to determine both structure and behaviour of the system and its GUI, in order to be used on the generation of a simulator (for both the GUI and system), and a GUI prototype application (in a specific platform) as well. However, the generated GUI prototype will only have the predefined entities that were explicitly modelled. For instance, if the system configuration changes, we would have to remake all of our interface model in order to adapt this changes. Also, we want to be able to give the user, the possibility to rapidly change visual paradigm (2D to 3D) and/or task modality [13],[12] according to his/her needs or operational context.

To explore the dynamicity of the interfaces, we approached the expressivity of the model transformation paradigm. Our aim now, is to develop an agent-adaptation description language (AADL) which is able to express (H)ALL-to-(H)ALL model transformations in runtime. Typically model transformation languages are composed of three essential sublanguages: the
pattern matching language, the transformation rules language and the transformation control flow language.

The pattern matching language is a query sublanguage of AADL that performs the matching/recognition of the running (H)ALL model entities. These query sentences express filters over the four kinds of (H)ALL model components (user, visual, task and system) and conditions over their existence, and over all the syntactic structures present in the (H)ALL metamodel (data structures, message handlers, etc.). Furthermore, we can associate variables within the query sentences which will be filled for each possible result of the query. These variables can then be used within the transformation rule sublanguage of AADL, where we are able to specify (according to a specific matching event) how and which (H)ALL components should be created, deleted and/or modified.

The control flow sublanguage of AADL is made by a state machine which models the behavioural aspects of the adaptation agent (AA). Typically, an adaptation agent has only two states (match and apply), and there may be several concurrent agents, accessing to the same (H)ALL running model, since a user may have installed several AAs on the running interface, or simply by the fact that a group agent is able to adapt any running interface in a certain user's group. Therefore, in order to guarantee adaptation correctness, the adaptation language semantics must assure some level of mutual exclusion protocol on these (H)ALL-to-(H)ALL model transformers. For instance, if the running (H)ALL model is currently locked at the moment of matching (probably by another AA), then the agent will just have to wait for another turn. We expect to introduce some expressivity about transactions in the future in order to enhance the AADL's control flow sublanguage.

With this language we expect again to not only be able to produce a simulation model of the AA descriptions, but also to produce runtime transformers to dynamically change the GUIs at runtime. Following the BATICCCS methodology, we are now specifying the AADL's abstract syntax and semantics, and then we will use this information to both generate an graphical editor for the language (possibly using the GMF metamodeling tool), and the ATL transformation language to implement the semantic correspondence of AADL syntactic structures with (possibly several) target languages syntactic structures which are able to automatically give them some computational meaning. The validation design of the AADL will involve the following activities:

- extraction of an AA model from the existing MyPanel prototype in order to express the loading of new system components based on an input model data source.
- extraction of an AA model from the existing MyPanel prototype in order to express the loading of new visual and task components based on the system components.
- definition of an AA model (and its application on the MyPanel prototype) which is able to express the transformation of the existing 2D interface into a 3D interface without matching any system component.

IV. RELIABLE CONTROL SYSTEMS

Verification and testing activities, in the context of an integrated development environment can be viewed as devel-
A. Static analysis

With the (H)ALL models we can now generate an analysis model which is suitable to be processed by model checking methods and tools. We are currently searching for the most suitable model checkers available (SPIN [7], NuSMV [1], etc.) to be integrated within our development’s framework. This integration involved the study of the most pertinent behavioural properties that the designers want to extract from a certain (H)ALL model, while validating its GUI.

We have found that most of the properties that designer’s might want to get verified from a certain (H)ALL model usually are composed by temporal connectives (expressed in a temporal logic) and are of three main categories:

- Reachability properties: ‘Whenever we push button A, the system simulator we will eventually reach state S’.
- Safety properties: ‘Whatever you do, we will never enter critical state without presenting a warning message in the GUI’.
- Liveness and Fairness: ‘It will always be possible to click an enabled button’.

We can see these safety and reachability properties as behavioural conformance statements between system components and graphical components. Obviously, the checking will have to take into account the structure and behaviour of each hierarchy (system and visual) and also the control communication performed by the affected task components. Furthermore, we can greatly restrict the state-space exploration by specifying which tasks we want to take into account in the GUI’s verification.

We expect that this development effort on a model checker integration will clarify on how the extraction of safety and reachability properties from the (H)ALL models (expressed in temporal logics) can be done, and how the designer’s interpretation of the results from the model checker can guide him/her to correct a certain specification error.

B. Dynamic analysis

Given a certain running (H)ALL model, at each time during runtime, it can be simulated and/or checked. Whenever this running (H)ALL model, is modified by an AA, we have to check it again for the specified quality properties (safety and reachability properties over states of the (H)ALL components). However, to provide a full verification of all characteristics of the GUI, we also need to be able to simulate and model check the specified AADL’s adaptation agents. This separation of dynamic and static analysis is important on one hand to help avoiding the state-space explosion (which most of the model checkers suffer) - in fact, on each version of the running (H)ALL model we are imposing a cone of influence reduction. On the other hand, the designer will be forced to clearly separate the quality properties that are most related to the dynamic aspects of the GUI model (creation and modification of (H)ALL entities), from the static aspects, which is only related with the running behaviour of a specific (H)ALL model. Controlling these two aspects is important in the integration of model checking and verification techniques within the BATICCCS framework.

C. Testing

We have proposed a full model based methodology the construction of reliable control systems. However, in practice, control systems are heterogeneous by nature, and depends on many legacy code. Therefore, it is important to ease the introduction of new external components. We propose an incremental approach by model refinement:

- we start from an incomplete, yet abstract conceptual model based on the designer’s knowledge of the external component - i.e a coarse understanding of the component’s structure, the internal behaviour and the external interface event-behaviour in the application context of the system under test.
- we then introduce this simplified model of the external component in our existing global (H)ALL+AADL model.
- We can then repeat the process by refining the simplified model to get more and more precise, i.e to model the external component’s behaviour with more accuracy. For instance, if it is known that the external component’s behaviour is internally creating new graphical entities, we can then express it with an AA’s model.

Our hypothesis is that if we view an external component as a black box, as we study its structure and behaviour, and continuously refine its computational model in (H)ALL+AADL, it turns out that it starts to become more and more manageable to simulation and model checking tools. Also, with this simplified model plus the external interface event-behaviour of the external component, we can semi-automatically generate test oracles [8], which can help the designer while refining and correcting its abstract model assumptions.

V. Conclusions

On this project, we brought a declarative description of both internal and external behaviour of processes, in order to provide simulation and testing capabilities. Also, we applied a clear design methodology having the subject of establishing a rationale over this particular control system and its user interface. Based on the source code analysis and a particular database configuration file, we extracted a description (expressed in (H)ALL) of the ATLAS-TDAQ control system and a small GUI, where we establish the behavioural relations between the entities of the system’s description with the entities present on its GUI.

The (H)ALL specification is a static representation of the structure and the behaviour of the domains, and this can simplify the model verification analysis. In order to produce
adaptive interfaces, we are currently developing an adaptation language with the capability to express the creation and update of new (H)ALL entities at runtime.

We aim to provide the capability to the express behavioural conformance between interface elements and the simulator nodes (in both static and dynamic specifications) by using task and message definitions. Provided that it is possible to automatically generate test cases from petri-net specifications [8], we will be able to do the same from both the system’s specification and property specification.

REFERENCES