A METHODOLOGY FOR CONTROL SYSTEMS GUI PROTOTYPING – A CASE STUDY*

M. Risoldi, D. Buchs, Université de Genève, Geneva, Switzerland
L. Masetti, CERN, Geneva, Switzerland
V. Amaral, B. Barroca, Universidade Nova de Lisboa, Caparica, Portugal

Abstract

Implementing Graphical User Interfaces (GUIs) for complex control systems (CS) implies many development challenges, especially for prototyping and refining. We propose to improve current practices by introducing a model-based, domain specific approach to GUI development. Our methodology is founded on the assumption that most information to be used for GUI prototyping can be derived by the CS specification itself. We use model transformation techniques for automatic generation of a GUI from a domain specific model. We apply the methodology to the CERN CMS Tracker Cosmic Rack as a case study.

INTRODUCTION

Modeling GUIs for control systems has requirements which are sometimes hardly met by general-purpose programming languages. The need to express domain features, together with the need of paradigms familiar to the domain experts, lead to the demand for domain specific languages (DSLs). While DSLs exist for specifying interactive user interfaces, our work exploits a few characteristics of the control systems domain to skip the GUI specification overall, while concentrating on describing the system to be controlled. At the same time, our method provides a system simulator, used for model checking and evaluation at times when the real system is not available. The methodology is centered on a DSL called Cospel (Control system SPEcification Language) and on model transformation techniques and is described in [4]. In this article we overview the methodology and apply it to a case study in the HEP field, the CERN CMS Tracker Cosmic Rack.

METHODOLOGY OVERVIEW

There are a few problems in building GUIs for complex control systems. Among them, a main one is that the people who actually know the system in detail (system engineers) are often not those who develop the GUI (software engineers). This introduces inefficiencies and potential lack of comprehension in GUI developers. This is even more a shame, as a lot of information needed for GUI development is already known by system engineers, who simply lack experience or time to translate that to a working GUI. We propose a development methodology to improve this situation. Resumed in Fig. 1, the methodology starts with gathering the existing information (1) about the system in a Cospel model (2). Using this language, a system engineer can use familiar concepts to describe the system he knows, not worrying about the interface. Models can be validated, and structural properties can be checked on them.

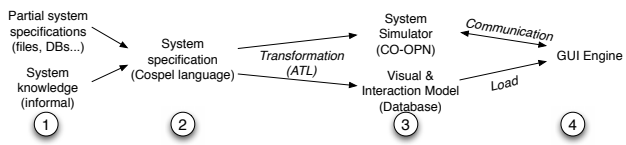

Figure 1: Methodology overview

In the third step of Fig. 1, the Cospel model is transformed to two artifacts: a visual and interaction model (made of all model information relevant to the GUI) and an executable system simulator. The latter is useful because for various reasons (cost, time, unfinished status) the real system may not be available for evaluating the GUI against a reactive system. The model transformation is an automatic, push-button operation requiring no special training or knowledge. The simulator can be used to verify behavioural properties of the model, as it has formally defined semantics including concurrency and transactionality. It is made in the CO-OPN language[1], from which in turn Java code is automatically generated.

The last step is the GUI generation. A GUI engine written in Java loads the visual model, and automatically gives a three-dimensional representation of it. One can navigate the system in a spatial way by moving the observer point around, or follow the system hierarchy for a structural navigation. Commands can be input by selecting objects. States are represented as object colors. The engine also loads the simulator and communicates with it via a driver, sending command and receiving events, so that the real experience of using the GUI with the system is simulated. At this point the software engineer can work on this already functional prototype to gather user feedback and adapt the GUI. At the end of the prototype stage the driver can be substituted with one for the real system.

CASE STUDY DESCRIPTION

The CMS experiment at CERN is a large particle detector installed along the Large Hadron Collider facility. Its Silicon Strip Tracker component is a complex system made of about 24000 silicon detectors, organized in 1944 *power groups*. These have several environmental and electric parameters to monitor. Tens of thousands of values and probes have to be controlled by the Tracker Control System[2]. For our case study, we took an early prototype

* Work supported by the Hasler Foundation of Switzerland and Portuguese FCT project PTDC/EIA/65798/2006

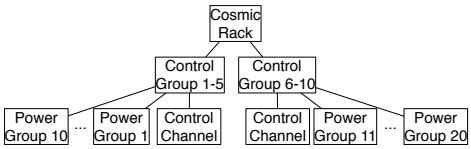

Figure 2: Cosmic Rack hierarchy

of the Silicon Strip tracker, called the Cosmic Rack (CR). This is equivalent to a section of the full tracker, mantaining the same hierarchical complexity, but with a reduced total number of components. The hierarchical structure of the CR is in Fig. 2. Each component is characterized by a finite state machine (FSM) with 4-8 states, and several transitions. There are rules stating how components change state according to the state of their sub-components. For each component we have a list of possible commands to send, and we know how these commands should affect the state of the component. Components also have properties, like temperatures or currents, and we have information about which threshold values should trigger alarms. Finally, we have the mechanical engineering information, telling us the shape, size and position in space of the components. Note that all this information is something one normally has already when treating a complex control system. They have not been created especially for the purpose of creating a GUI, but are simply structural, safety and input/output features of the system.

MODELING THE COSMIC RACK

Data about the system was already present in an electronically-usable format, and it would have been possible to process it automatically to generate the model. However, for the sake of this study we created a model of the CR from scratch, pretending that the existing information is only available in non-electronic format.

The main concepts used in the Cospel language are those common to the control systems domain. A *System* is made of *Objects*, which are its individual components. Each Object has a *Type*, which describes features common to all objects of that type. Separating these two concepts helps reusability and agility of specification. Types are associated to *FSMs*, which are made of *States* and *Transitions*. Types also describe *Properties* with threshold values for alarms; *Commands* and *Events*; and geometrical *Shape* of objects. Objects on the other hand have a *Position* in space, and hold information about the hierarchy (an object can have one *Parent* and/or several *Children*). Note that also types have hierarchical information: according to the structure in Fig. 2, the Control Group type states that its children are one Control Channel and ten Power Groups. This is used to guide and validate the hierarchy of objects, who have to respect this template.

As Cospel has been defined using the ECore formalism[3], it was possible to automatically generate a simple editor for it using Eclipse[1]. Fig. 3 shows a screenshot of the editor, containing a partial specification. We can see the object *CG6-10* (one of the two control groups), its type *CTRL-GRP*, and its FSM *FSMControlGroup*. The object, selected, shows its properties in the bottom panel. Namely, we see its association with its type, its parent (not on screen in the figure) and its name. It also contains its absolute coordinates in space.

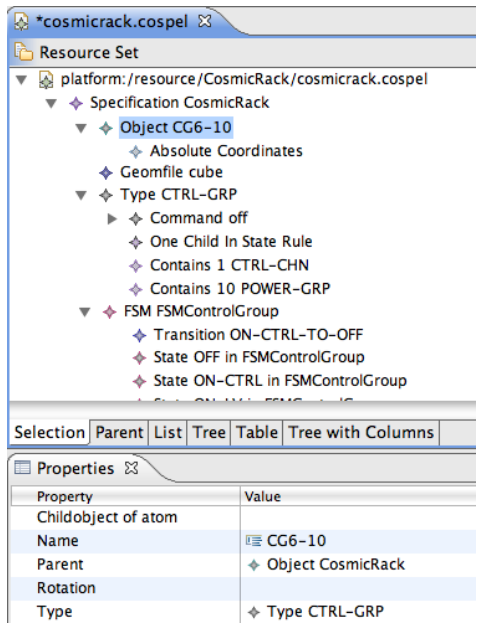

Figure 3: Cospel editor: a partial specification

We see the type has hierarchy templates (*Contains...*). It also has a rule (*One Child...*), saying how objects of this type must change state when one of their children goes to a given state (e.g. if one child goes to error, the object also should go to error). It is possible to define rules of kind *One child is in state* or *All children are in state*. We are currently working on defining rules based on percentages of children in a given state.

The screenshot also contains an FSM called *FSMControlGroup*, with the related States and Transitions.

Associations between elements of the language can simply be made via the property panel. Copy and paste, as well as drag and drop, are supported to make large arrays of objects, or reuse pieces of specification. It is possible to validate the specification directly in the editor, to know if it is coherent and complete.

Note that, while this automatically generated editor is functional, it is possible to create richer and more usable editors. In particular, techniques like GMF can be used to give a visual syntax to parts of the specification like FSMs (which would speed operations up noticeably). So there is still space for improvement.

CREATING THE SYSTEM SIMULATOR

Once the specification of the system is complete, we can use it for generating its formal executable model. This is achieved as a push-button operation, which takes the

[1] http://www.eclipse.org

Cospel specification and inputs it to an ATL[2] transformation library. This library is a set of declarative rules, matching each element of the Cospel model and creating a corresponding element of the CO-OPN model. The CO-OPN model is then transformed into executable Java code. This gives the possibility of simulating the system response to the GUI. Information used for the system simulation is the system's hierarchy and complete behavioural information (FSMs, commands, events, properties, rules). Note that the simulator generation is optional: if the user only wants the GUI prototype, he can choose to just generate that, and skip modeling all the behavioural part.

CREATING THE VISUAL MODEL

The choice of what to have in the visual model was driven by a few assumptions. First, if we want to monitor an apparatus, it would be reasonable to have the GUI visualize the system's state. Second, if components must receive certain commands, the GUI should provide means to send them. Third, if the system is complex it would help to have some structured representation of it. Fourth, as some issues with control systems are spatially-related (e.g. overheating), it would be helpful if the system representation was spatial. For these reasons, information used for the visual model is the system's hierarchy, the declarations only of states, commands, events and properties, and geometrical data. The visual model is stored in a database, and again is obtained via a push-button transformation of the Cospel model to SQL queries.

GUI PROTOTYPE

The GUI prototype is created by a GUI engine (written in Java) which loads the visual model and uses it to show the system under control. We chose to use a three-dimensional representation, as we are also exploring the contribution of spatial perception to navigation in complex systems. Note that other types of interfaces are also possible, the most obvious being a tree-like visualization of the system (a quite common metaphor in the field).

Users can move in the 3D scene and navigate levels of the hierarchy. They can click objects to investigate their parameters, and send them commands. Object states are represented as colours; this gives an at-a-glance understanding of the overall system situation. At all times a tree with the system hierarchy is available as a quick way to move to a given object. A stereoscopic visualization mode is available if immersion is desired.

When loading the visual model, users can also choose to load the system simulator (if it was generated). This initializes a communication driver which instantiates the simulator; it routes commands from the GUI to the system simulator, and events in the other direction. One can thus send a command, and see the system state change following state change events from the simulator. At any time, the communication driver can be substituted by a custom driver for interfacing the actual system under control. One simply needs to fill a pre-made code skeleton with the correct procedures for communicating with the system.

The result for the CR GUI prototyping process is shown in Fig. 4. The visualization is at the level of Power Groups (the long, rectangular structures). Object colors represent states. The currently selected Power Group (PG_Layer_3_Rod_2) is showing its property panel, with buttons (off, on_lv, ...) for sending commands to the object.

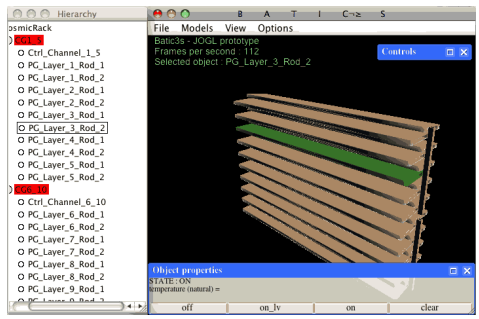

Figure 4: The GUI for the Cosmic Rack

CONCLUSION

We presented a methodology for automatically generating a Control System GUI prototype from pre-existing information about the system. This use case confirmed applicability of the approach to a complex control system, and the feasibility of GUI generation. The next steps of this work will focus on scaling to a larger system (the full CMS tracker) and improving ergonomics of the generated GUI.

REFERENCES

[1] O. Biberstein, D. Buchs, and N. Guelfi. Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha, F. D. Cindio, and G. Rozenberg, editors, *Advances in Petri Nets on Object-Orientation*, LNCS, pages 70–127. Springer-Verlag, 2001.

[2] A. Dierlamm, G. H. Dirkes, M. Fahrer, M. Frey, F. Hartmann, L. Masetti, O. Militaru, S. Y. Shah, R. Stringer, and A. Tsirou. The CMS tracker control system. *Journal of Physics: Conference Series*, 119(2):022019 (9pp), 2008.

[3] W. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM RedBooks, February 2004.

[4] M. Risoldi and D. Buchs. A domain specific language and methodology for control systems gui specification, verification and prototyping. In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007), 23-27 September 2007, USA*, pages 179–182. IEEE Computer Society, 2007.

[2] http://www.eclipse.org/m2m/atl/