A domain specific language and methodology for control systems GUI specification, verification and prototyping

RISOLDI, Matteo, BUCHS, Didier

Abstract

A work-in-progress domain-specific language and methodology for modeling complex control systems GUIs is presented. MDA techniques are applied for language design and verification, simulation and prototyping.

Reference


DOI: 10.1109/VLHCC.2007.21
A domain specific language and methodology for control systems GUI specification, verification and prototyping

Matteo Risoldi and Didier Buchs
Université de Genève
CUI Battelle - Bat. A, 1227 Carouge, Switzerland
{matteo.risoldi, didier.buchs}@cui.unige.ch

Abstract

A work-in-progress domain-specific language and methodology for modeling complex control systems GUIs is presented. MDA techniques are applied for language design and verification, simulation and prototyping.

1. Introduction

Modeling GUIs for control systems has requirements which are sometimes hardly met by general-purpose modeling 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).

We propose a methodology to develop complex control systems GUIs through the specification of the system under control. The methodology is centered on a DSL, Cospel, and is integrated in a formal framework allowing model checking, verification of implementation and prototyping.

This project is funded by the Hasler Foundation.

2. The domain of complex control systems

Control systems can be defined as mechanisms that provide output variables of a system by manipulating the inputs. While some systems can be very simple (e.g. a thermostat) and pose little to no problem to modeling using general-purpose formalisms, others can be complex with respect to number of components, dimensions, physical and functional organization and supervision issues.

A key feature of control systems is a hierarchical organization - an object can be composed of several sub-objects, in a tree-like, container-contained relationship. Objects at all levels can receive commands and generate events and alarms. Also, an object will have a state indicating malfunctioning or warnings. The state can depend on the object’s properties, and/or on the states of its sub-objects.

Figure 1. An example of hierarchical control system: drink vending machine

Figure 1 shows a partial example of such a system, a simple drink vending machine (DVM).

A language for modeling a GUI for such a system requires [2] expressing the following aspects:

- behavioural aspects of the system and GUI
- possibility of validating the model
- prototyping and verification of the GUI

3. The Cospel language

The first element in our methodology is a domain specific language (DSL), named Cospel (COnrol systems SPEcification Language). There are previous works on DSLs for control systems [14] and user interfaces [12], some of them model-based. These approaches however do not go much in the direction of formalization, focusing rather on engineering aspects, paradigms or other domain-specific issues. Our approach is more focused on formalizing the structure and behaviour of the system.
We chose the Model Driven Architecture (MDA) for DSL design for the following reasons: stability and non-ambiguousness of the meta-modeling formalism; possibility of meta-model based transformations on DSL specifications; flexibility of the metamodel for future modifications of the language concepts.

3.1 Motivations for a DSL

We define a DSL, rather than use a general purpose formalism, to make it possible for the system experts themselves (who don’t necessarily master complex general-purpose formalisms like UML) to give a specification of the system. A DSL can offer a more familiar view on the system, including the correct abstractions and hiding unnecessary complexity. It can however reuse some concepts from other widely accepted languages when these are familiar to the domain experts.

One recurring question by experts was, how is it possible to assure that the GUI behaviour for a control system is consistent with the system? The answer to this is that if the GUI behaviour is directly deduced from the system specification, this consistency can be assured.

3.2 Domain concepts and meta-model

The scope of the Cospel language is modeling a physical system with respect to its control aspects, and with the final perspective of prototyping a GUI for the system from this specification. We identified the main elements of the domain that we want our language to model as follows:

- Hierarchical composition of objects: we want to express systems having several levels of hierarchy, with objects being composed (physically and/or functionally) of subobjects.

- States and state transitions for objects: objects of the system have states, and transitions between them can be triggered and/or trigger events, such as notifications or alarms. Domain experts seem familiar with the finite state machines (FSM) formalism.

- Properties, events and methods of objects: objects have typed properties (e.g. temperature), events (e.g. alarms) and methods (e.g. a reset command). This is the base to be able to control them via a GUI.

- Rules for state calculation based on events, properties and/or subobjects’ states: property changes, state transition of subobjects or other events can trigger a state transition for a given object (e.g. it might go to a Warning state when one of its subobjects is in an Error state). This is definable through rules which can be eventually composed.

We give a meta-model of Cospel using the Eclipse Modeling Framework (EMF) [6]. The language concepts and relations are described with OMG XML Metadata Interchange (XMI). The complete meta-model is quite large and would not fit in this article. Figure 2 shows a small part of it, the definition of the FSM for objects, in order to illustrate the formalism used. An Object, part of a Specification, is associated to a FSM. The latter is comprised of one or more States and several Transitions, each with a to and from state. Note how FSMs are not an attribute of the object but are independent entities; this allows using instances of the same FSM for several objects which behave similarly - reutilization is a crucial point when modeling large scale systems. We followed this strategy whenever defining something which would be applicable to a large number of objects at one time.

Note the association of Object with itself, expressing the hierarchical composition of objects. Also note how associations have client/supplier roles explicitly named; these names are used in the EMF generated code to navigate the model.

Constraints can be expressed in the meta-model by means of Object Constraint Language [1] or by constraint enforcement at the validation stage.

![Figure 2. Partial meta-model: definition of FSMs for objects](image)
Based on the meta-model, EMF generates a set of Java APIs for creating and parsing models, as well as a simplified model editor, shown in Figure 3. More powerful editors, with facilities making use of domain specific abstractions, can be created using the Java APIs, which is part of the future work.

4. Adding semantics by model transformation

The meta-model gives Cospel an abstract syntax; what needs to be specified next is its semantics. In order to give it, we apply an MDA technique, giving semantics via meta-model based transformation. We have a specification in Cospel and its meta-model. If we transform it into another language of which we have a meta-model, and which has already defined semantics, we achieve the goal. The transformation can be defined once and for all by establishing transformation rules between the two meta-models. Figure 4 illustrates the procedure.

4.1 Choosing a target language: CO-OPN

Ideally, the target language should have features to let us do model validation, prototype generation and verification. The Concurrent Object-Oriented Petri Nets (CO-OPN) [4, 5] is a good candidate for this. It is a formalism based on algebraic Petri nets which features object orientation, true concurrency and algebraic data types. Its semantics have a rigorous formal definition [3] which allows for formal verification of properties. Finally, it has a Java prototype generator, so prototyping and testing of specifications can be performed. The feasibility of a control system model in CO-OPN has been explored in [13]. Also, CO-OPN meta-model based transformation has been explored in [11].

4.2 Transformation rules

Following is the list of rules to transform the main Cospel elements into corresponding CO-OPN elements, in the form Cospel element → CO-OPN element.

- **Object** → **Class** + **Context** (the Context is where the Class is instantiated, and provides synchronization with other classes)
- **Methods** → **Methods of the Class**
- **Properties** → **Places in the Petri net of the Class**
- **Data types** → **Algebraic Abstract Data Types (ADT)**
- **FSM** → **Places for states, a set of axioms for transitions**
- **Hierarchy of objects** → **Composition of Contexts within other Contexts (using a generic pattern which allows command and event routing between levels of the hierarchy)**
- **Behavioural part (event triggering)** → **Axioms in Contexts/Classes (e.g. connecting property changes to state transitions, transitions to events, events to methods)**
- **Composition rules (for states and properties)** → **Axioms in Contexts/Classes**

The transformation is implemented using the Java APIs generated by EMF from both the Cospel and CO-OPN metamodels.

4.3 The result of the transformation

A detailed description of the CO-OPN model resulting from the transformation is lengthy and out of the scope of this article, so only its structure will be illustrated. Figure 5 shows part of hierarchy of contexts for the DVM example of Figure 1, using CO-OPN graphical notation. The **DVM object has been transformed to a DVM class (light gray box in the center)** and a **DVMContext context (outer box)**. **Switch and a Reset methods of the DVM object have been translated to methods** (small black rectangles) of the **DVM class**. **Also, temp and a state events of the DVM object have been translated to gates** (small white rectangles) of the **DVM class**. **Children objects Fridge-01 and**
Hfigure 5. Hierarchy of the DVM example

Fridge-02 have been transformed into similar structures, and their contexts (dark gray boxes) have been nested inside DVMContext. They contain their own objects, and contain in their turn more contexts of children objects. All contexts have a method \texttt{Cmd} and a gate \texttt{Return} generated by the transformation, which allows for command and event routing among levels of the hierarchy. The behaviour of \texttt{Cmd} and \texttt{Return} is defined by axioms (represented graphically by arrows, but described by algebraic formulas in CO-OPN), which determine the routing of commands and events. The internal Petri nets of the classes (not shown) implement other class features (FSMs, properties...).

4.4 Verification

Verification can be performed on the model by several approaches:

- generating the state space for the model[3], we can verify properties expressed in Computational Tree Logic (CTL);
- test intentions can be generated from the model[9];
- a Java executable prototype of the model can be generated and used for simulation[7].

5. GUI Prototyping

Similarly to previous approaches for 2D database GUI generation[8], we make use of data types for deciding which kind of actuators to show in the GUI. We also take advantage of the geometrical information in the model to experiment a 3D stereoscopic GUI approach. A GUI prototyping engine[10], written in Java as well, takes care of building a 3D scene using the geometrical data, setting up interaction according to the available inputs/outputs for each object, and managing the coherence between the state of the system and the state of the interface. The result is a GUI that shows our system in 3D, lets us send commands and receive events, and represents the state of the system via color codes.

6. Future work

Current objectives of this project include:

- add the possibility of defining geometrical relationships between objects in a semantical way (e.g. "parallel to")
- case studies: CERN CMS experiment Silicon Strip Tracker control; computer room control

References