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

Matteo Risoldi and Didier Buchs
Centre Universitaire D’Informatique
Université de Genève

Abstract:
We present a domain specific language for modeling complex control systems user interfaces, defined by a meta-model and with semantics given by means of meta-model based transformation to the CO-OPN formalism. This language allows for easier specification by domain experts, verification and simulation activities by software engineers, and prototype generation. An example of specification is shown, as well as verification on the obtained model. An overview of the prototyping architecture is given.
Contents

1 Introduction 2

2 The domain of complex control systems 2

3 Cospel language 3
   3.1 Motivations for a DSL 3
   3.2 The meta-model 4
   3.3 An example model 6

4 Adding semantics by model transformation 7
   4.1 Choosing a target language: CO-OPN 7
   4.2 How the transformation is done 8
   4.3 The result of the transformation 8

5 Prototyping 9

6 Property verification 10
   6.1 Verification with CTL 10
   6.2 Verification by simulation 10

7 Summary and future work 11

8 Acknowledgments 12
1 Introduction

Modeling user interfaces for the domain of control systems, as many other specific domains, has requirements and challenges which are sometimes hardly met by standard, general-purpose modeling languages. The need to express domain features, together with the desire to express them using paradigms familiar to the domain experts, leads to the demand for domain specific languages. However, when creating a language many different approaches can be followed, from the highly imperative to the extremely functional, from the operational semantics to formal definitions, basically depending on the field of application, type of users and purpose of the language.

We propose a methodology to develop complex control systems user interfaces through the specification of the system under control. The methodology is centered on a language, Cospel, which is specific for modeling control systems, and is integrated in a formal framework allowing model checking, verification of implementation and prototyping. In the following sections we will describe the domain and language. Then we will discuss the semantics definition, and we will illustrate property verification and prototyping through examples. Conclusions and a future work overview will wrap up the article.

2 The domain of complex control systems

Control systems (CS) can be defined as mechanisms that provide output variables of a system by manipulating the inputs (from sensors). While some CS 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 complex control system will generally have a hierarchical organization, in which every elementary object can be grouped with others, and composite objects (groups) can be grouped as well, forming a hierarchical tree in which the root represents the whole system and the leaves are the most elementary devices forming it. Typically this grouping will reflect a physical container-contained composition (e.g. a fridge contains several compartments with different temperatures), but it could follow relations of other nature, functional for example. Elementary and composite objects can receive commands and communicate states and alarms. It is generally the case that the state of an object will depend both on its own properties and on the states of subobjects forming it. Operators can access this hierarchy at different levels of granularity. Figure 1 shows a partial example of such a system, a simple drink vending machine (DVM). We will use it also in following examples.

The development of a graphical user interface (GUI) for such a system brings many requirements when the methodology to be used is a formal one. First, there has to be a language which lets the developer model the behavioural aspects of the system under control (and of the GUI). Then there has to be a methodology which allows for validation of the model, prototyping of the GUI and verification of the prototype. Our approach sees the interface as tightly coupled to the system controlled, and aims for its development through the description of the system under control. From this
system model, which includes information needed for GUI development, we generate a GUI prototype using model transformation and code generation techniques.

3 Cospel language

The first element in the methodology is the domain specific language (DSL), named Cospel (COntrol systems SPEcification Language). There are previous works on DSLs for control systems [17] and user interfaces [14], some of them model-based. These approaches however do not go very 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 [11] approach to designing the DSL for a few reasons. Firstly, the definition of a DSL through a meta-model expressed in a UML-like formalism facilitates the task of defining the elements of the language by giving a clear schematics for them. Secondly, using a stable and non-ambiguous formalism ensures interpretability of the meta-model by different application (or by future users). Thirdly, it is possible to transform a DSL expressed by a meta-model into other languages by defining meta-model based transformations.

3.1 Motivations for a DSL

The reasons for choosing to define a DSL rather than use a general purpose formalism come from the goal we are pursuing. We want the GUI to be built from a specification of the system to be done by actual system experts, not by software engineers. In the survey we made in the preparation phase of the project, we felt there was a gap between the understanding of the system that experts had, and the vision software engineers had. One of the most relevant differences was that system experts could not really "see" the system behaviour in the interface in the way they wanted. We

Figure 1: An example of hierarchical control system: drink vending machine
want to make it possible for the system experts themselves to give a specification of
the GUI.

Also, one recurring question by experts was: is the GUI behaviour going to be
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.

However, specifying a complex control system presents a number of challenges.
There are a large number of components, and their liaisons are often complex with
respect to spatial relationships and functional dependencies. While general purpose
formalisms can be used to model these systems, the model risks being both very
large and very difficult to understand. Also, the system experts who are supposed
to give the specification are often not very experienced in using complex abstract
formalisms like Statecharts or UML. A DSL, by presenting a familiar view on the
specification, can offer a clearer view to these users as to what to specify and how
to do it. Also, by determining the correct scope and elements of the language and
making the correct abstractions, it is possible to empower this language with tools,
such as an editor, which can simplify complex tasks, like applying a parameter to a
large number of components or making queries based on spatial relationships.

3.2 The meta-model

We give a meta-model of Cospel using the Eclipse Modeling Framework (EMF)
[6]. The language concepts and relations are described with a subset of a UML
class diagram; this representation is then converted into a OMG XML Metadata
Interchange (XMI) file conforming to the EMF Ecore model. From this XMI, Java
code is generated that can be used to write and parse models expressed in the
language, as well as a simple editor to create them. We will see later how this Java
code is used for transformation.

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. After a survey of the domain during the preparatory phase
of this project (at CERN laboratories, Geneva) 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 function-
ally) of subobjects, which are composed in their turn etc.

- States and state transitions for objects (as finite state machines - FSMs):
  objects of the system must be characterized by states, and transitions between
  them can be triggered and/or trigger events, such as notifications or alarms.

- Properties, events and methods of objects: it must be possible to characterize
  objects with 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 type of events
must be able to 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 must be definable through rules which can be eventually composed.

- Geometrical information about objects: we want the physical model of the system to be included for a twofold reason. First, spatial information is critical to alarms such as temperature/humidity warnings (a fire in an object can propagate to neighbouring objects). Second, it allows us to give a visual representation of the system in a GUI, which is one of our research themes (as discussed in the final section).

The complete meta-model is quite large and would not fit in this article; we can show parts of it to give an idea of how it looks like. Figure 2 shows the definition of the FSM for objects. 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.

Figure 3 shows the rules to calculate an object’s state based on the state of its subobjects. Here the Object may contain several Statecomprule. Each of these is associated to a resulting State of the object (this is the same State class of Figure 2), which is attained if a Condition is verified. This can be either an Atom, or a conjunction of several Conditions (via And, Or and Not operators). The Atom verifies (through its associations) if a given sub-object (another Object) is in a given State.

Note in both figures the association of Object with itself, expressing the hierarchical composition of objects. Also note how associations have client/supplier roles explicitly named; these names will be used in the EMF generated code to navigate the model (some names have been omitted in Figure 3 for readability).

A remark that can be done is that there are no constraints expressed in this meta-model. For example, nothing says that the association of Atom with Object in Figure 3 has to be with a subobject of the Object associated with the corresponding Statecomprule - one could put any object there. This limitation comes from the fact that EMF does not currently work with constraint languages such as OCL. There are projects [2] to include OCL support in EMF, and we aim at integrating these capabilities in our methodology, but until then constraint checking has to be a hard-coded part of model validation.

The main goal with respect to editing tools for Cospel is to simplify the repetitiveness of tasks induced by system complexity, like defining the same parameters for a large number of objects, or creating series of objects. By using some domain specific features, like the fact that generally large numbers of objects can be gathered in the same class with common features, or that physical arrangement of objects has often some symmetries, we can empower our editor to make the best possible use of the domain concepts expressed in the DSL. We did not yet develop a specific editor for our Cospel. However we can show a simple example model, made with the Eclipse automatically-generated editor (another advantage of MDA - defining the meta-model of a language gives all the information to edit models in that language).

### 3.3 An example model

After exporting said meta-model to XMI, we can use the Eclipse EMF plugin to automatically generate Java code to edit and navigate models expressed in Cospel. This code can be used to implement advanced editors without having to implement all the model-management part; however, Eclipse also generates a basic editor for quick prototyping in the language we have defined. This editor lets us create and edit objects in a tree view, according to the meta-model we gave.

For the sake of an example, we want to give a model which relates to the general problem, by presenting the features we discussed earlier - hierarchy of objects, finite state machines for objects, events triggered by state dependency rules, geometrical information, etc. The only feature of the system we simplify on is cardinality - we only give a few objects. We consider a safe assumption that once the language is able to model the complexity of the system, the main challenge added by cardinality would be repetitiveness, and as said this is a challenge we intend to tackle via the
Adding semantics by model transformation

What is yet missing from the Cospel specification, to be able to generate prototypes from it, is semantics. In order to give it, we apply a technique that MDA offers, giving semantics via meta-model based transformation. We have a specification in Cospel and its meta-model. Now, if we could transform it into another language of which we have a meta-model, and which has formally defined semantics, we could achieve the goal. The transformation can be defined once and for all by establishing transformation rules between the two meta-models. Figure 6 illustrates the procedure.

Choosing a target language: CO-OPN

Ideally, the target language should have such features to let us do model validation, prototype generation and verification. For these reasons, we chose to transform our Cospel specifications to Concurrent Object-Oriented Petri Nets (CO-OPN) [4, 5]. This is a formalism and language based on algebraic Petri nets which features object orientation, true concurrency and algebraic data types. Moreover, its semantics have a strong formal definition [3] which as we will see allows for formal verification of properties. Finally, it has a prototype generator, so it also will let us perform
prototype testing. The feasibility of a control systems model in CO-OPN has been explored in [16]. Also, CO-OPN meta-model based transformations has been explored in [13].

4.2 How the transformation is done

Table 1 lists the rules to transform main Cospel elements into corresponding CO-OPN elements.

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

Table 1: Transformation rules

Some generic patterns have been defined in the transformation rules to have a consistent CO-OPN specification from the Cospel model.

The transformation is accomplished using the Java code generated by Eclipse from both the Cospel and CO-OPN EMF metamodels. This code, comprised of all accessor and writer methods needed to parse and create models in the respective language, is called by human written code that implements the transformation.

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 we will just illustrate its structural. Figure 7 shows the hierarchy of contexts for the DVM example, using CO-OPN graphical notation. You can see how the dvm object has been transformed to a dvm class (bright gray box in the center) and a DvmContext context (outer box). You can see how a Switch and a Reset methods of the dvm object have been translated to methods (small black rectangles) of the dvm class. Also, a temp and a state events of the dvm object have been translated to gates (small white rectangles) of the dvm class.
Section 5 Prototyping

The children objects Fdg0001 and Fdg0002 have been transformed in their turn in similar structures, and you can see how their contexts (dark gray boxes) have been nested inside DvmContext. They contain their own objects, and may contain in their turn more context of children objects. All contexts have a method Cmd and a gate Return; this is an artifact generated by the transformation, and allows for command and event routing among levels of the hierarchy. The behaviour of Cmd and Return is defined by axioms (represented graphically by arrows, but described by algebraic formulas in CO-OPN), which determine the correct routing of commands and events.

5 Prototyping

The CO-OPN tool package includes a code generation feature. This creates an executable Java implementation of a CO-OPN specification. Java classes are created for CO-OPN classes and ADTs, and implement CO-OPN methods and events. Normally, this code is meant for simulation running inside the CO-OPN editor. However, it is possible to import this code into a larger code base to use it as a basis for prototyping the system GUI.

Since the model we based all the methodology on has been built by modeling aspects which are relevant to GUI development, and the language has been expressly designed for it, we can safely say that the information we need for GUI prototyping is present in the model. The semantics of controls for every object is present under the form of the semantics of communication with system objects in the model. Moreover, since we modeled the objects behaviour, we have at the same time a limited but functional simulator of the system under control. Starting from this code base, there are several ways to implement a GUI prototype. Some approaches apply visual paradigms based on the semantics of the objects under control, and for example choose to represent a given object by a button, rather than a dial or a slider, based on the range of values of the inputs. We choose to integrate into the model the information for visual representation, as we are oriented to GUIs which show the system in a 3D stereoscopic view. This is the reason why the meta-model includes, for every object, information about its geometry and position. This information is used by a GUI prototyping engine, written in Java as well, which 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 as we defined it, lets us send commands and receive events, and represents the state of the system via color codes. A screenshot of this prototype is shown in Figure 8 for a somewhat more complex DVM than the one we discussed. Although difficult to appreciate in black and white, objects are shown with different colors according to their state.
6 Property verification

As with many formalisms, there are several ways of performing verification of properties on a CO-OPN specification. As an example, we will see how this can be done with Computational Tree Logic (CTL)[8] and with simulation. The former allows verification of a property on the state space of the system. The latter allows for verification of feasibility of given sequences of operations satisfying a property.

The property we will verify on the DVM example of the previous sections is the following: when the object Fdg0001 goes to state Error, the object dvm will always eventually end up in state Error.

6.1 Verification with CTL

The property can be expressed as follows in CTL:

\[ P = AG (Fdg0001.state = \text{Error} \rightarrow AF \ dvm.state = \text{Error} ) \]

meaning: it is always true (AG) that, if Fdg0001 is in state Error, dvm will always finally (AF) be in state Error. We can rewrite the property in minimal CTL syntax with only EU (exists until), EG (exists globally), ∧ and ¬ operators, as

\[ P = \neg(true EU(Fdg0001.state = \text{Error} \land EG \neg dvm.state = \text{Error}) ) \]

which says that there is no state in which Fdg0001 is in error and there are paths to have dvm in a non-error state.

To verify it, first we generate the transition system for the CO-OPN specification (this is possible thanks to the formal definition of CO-OPN semantics [3, 15]). For the sake of simplicity and readability, let’s suppose that the state of objects is the only property of the system, so that, with 3 objects and 3 possible states for each, we have 27 possible states. A partial transition system is visible in Figure 9, where state name \( s = s1s2s3 \) denotes that dvm, Fdg0001 and Fdg0002 are respectively in states \( s1, s2 \) and \( s3 \), and where the object states OK, Warning and Error are abbreviated in O, W and E. The thicker arc shows the transition corresponding to the Statecompositionrule forerror of Figure 5.

By using standard algorithm for CTL model checking [9], it is easy to evaluate \( P \) on the states and see if it holds.

6.2 Verification by simulation

The code generated from the CO-OPN specification can be used to perform execution of input sequences to verify if the behaviour matches the intended specification. The CO-OPN environment offers a simulation engine to do this, allowing the user to call methods, and returning information about events in the system.

For instance, to verify our property we set Fdg0001 to state Error (or give sequences of input that result in this state), and we verify if the simulator notifies
us with the expected state for the dvm object. The observed result of the method call generated by the simulator is something in the form:

\[
Fdg0001Context.Cmd (Fdg0001, \text{setState Error}) \text{ with } \\
Fdg0001Context.Return (Fdg0001, \text{state Error}) // \\
DvmContext.Return (dvm, \text{state Error})
\]

The *with* separates the input (the method we called) from the output (the events in the system). In this case, our *setState Error* method of *Fdg0001* activated its FSM, and the *Statecompositionrule forerror* of the *dvm* object. The output communicates that both *Fdg0001* and *dvm* changed their state to *Error* simultaneously. This simultaneity is expressed by the // operator.

Of course, identifying the relevant sequences of input to provide presents theoretical and practical challenges. Work is ongoing [12] for generating test suits from CO-OPN specifications semi-automatically.

7 Summary and future work

Using model-driven architecture, we designed a domain specific language for control systems. This language implements characteristics which can be used to reduce the complexity of modeling the system user interface, by using known information about the system to deduce a prototype of the GUI. The features of the language are meant to allow the control systems expert to specify a highly complex system more rapidly. By means of model transformation, we can achieve a semantically defined, executable specification on which to perform formal verification, testing and prototyping.

Current lines of work in this project are oriented to including in the meta-model more GUI-specific information. One of the aspects we want to be able to express in the Cospel language is everything related to user profiles and task modeling. This will have to be expressed in such a way that it makes use of concept familiar to control systems experts, and will be used to have GUIs which can adapt to the user, and/or guide him through pre-defined workflows for certain tasks. Another aspect we are working on is enriching the semantics of the geometrical information for the objects, by giving the language the possibility of defining an object position not only in an explicit way (e.g. xyz coordinates), but as relationships with other objects (e.g. saying "object a, b and c are in a row at a distance x from each other").

For what concerns case studies, we are collaborating in this study with the CERN laboratories in Geneva (CH) in the context of the CMS Silicon Strip Tracker. This is a large and complex sub-atomic particle detector, used in a high-energy physics experiment called CMS. It is made of about 15,000 components [1], with strict requirements for safety control. For its size and complexity it makes an excellent candidate for a case study. We are collaborating with researchers at CERN who are involved into visualizing the state of the Tracker in 3D, and in the following phases of the project will have the results of our methodology evaluated by users of the control interfaces at the CMS experiment.
We are also studying if the features we identified for the control systems domain can be extended to other near domains, so as to find other applications for this methodology. The main lead is examining how the geometrical information can be used to express non-spatial relationships, like functional ones. This would allow to represent control systems which might be less physical and more logical in nature, like for example networks which might be represented visually by their topology.

8 Acknowledgments

This article has been written in the context of the BATIC$^3$S project, funded by the Hasler foundation grant MMI 2003. We would like to thank the other participating institutions: Ecole d’Ingénieurs de Genève, Haute Ecole Valaisanne, CERN, Nova Universidade de Lisboa.

References


REFERENCES


Figure 3: Partial meta-model: definition of state composition rules
Figure 4: Eclipse-generated model editor
Figure 5: Pre- (left) and Post-state (right) of the Statecompositionrule for error

Figure 6: Schema of meta-model based transformation
Figure 7: Hierarchy of the DVM example
Figure 8: Screenshot of the GUI prototype

Figure 9: Partial transition system for DVM CO-OPN specification
SMV Technical Reports

This report is in the series of SMV technical reports. The series editor is Didier Buchs (Didier.Buchs@unige.ch).

Any views or opinions contained in this report are solely those of the author, and do not necessarily represent those of SMV group, unless specifically stated and the sender is authorized to do so.

You may order copies of the SMV technical reports from the corresponding author or the series editor. Most of the reports can also be found on the web pages of the SMV group (http://smv.unige.ch/).