A language and a methodology for prototyping user interfaces for control systems

Risoldi, Matteo; Amaral, Vasco; Barroca, Bruno; Bazargan, Kaveh; Buchs, Didier; Cretton, Fabian; Falquet, Gilles; Le Calvé, Anne; Malandain, Stéphane; Zoss, Pierrick

How to cite

RISOLDI, Matteo et al. A language and a methodology for prototyping user interfaces for control systems. In: Human Machine Interaction. Research results of the MMI program. Berlin : Springer, 2009. p. 221–248. (LNCS) doi: 10.1007/978-3-642-00437-7_9

This publication URL: https://archive-ouverte.unige.ch//unige:5104
Publication DOI: 10.1007/978-3-642-00437-7_9

A language and a methodology for prototyping user interfaces for control systems

Matteo Risoldi[2], Vasco Amaral[1], Bruno Barroca[1], Kaveh Bazargan[2], Didier Buchs[2], Fabian Cretton[3], Gilles Falquet[2], Anne Le Calvé[3], Stéphane Malandain[4], and Pierrick Zoss[4]

[1] Universidade Nova de Lisboa, vasco.amaral@di.fct.unl.pt, mailbrunob@gmail.com
[2] Université de Genève, {kaveh.bazargan, didier.buchs, gilles.falquet, matteo.risoldi}@unige.ch
[3] HES-SO Valais, {anne.lecalve, fabian.cretton}@hevs.ch
[4] Ecole d'ingénieurs de Genève, {stephane.malandain, pierrick.zoss}@hesge.ch

Abstract. The BATIC[3]S project[5] (Building Adaptive Three-dimensional Interfaces for Controlling Complex Control Systems) proposes a methodology to prototype adaptive graphical user interfaces (GUI) for control systems. We present a domain specific language for the control systems domain, including useful and understandable abstractions for domain experts. This is coupled with a methodology for validation, verification and automatic GUI prototype generation. The methodology is centered on metamodel-based techniques and model transformations, and its foundations rely on formal models. Our approach is based on the assumption that a GUI can be induced from the characteristics of the system to control.

1 Introduction

Modeling user interfaces for the domain of control systems has requirements and challenges which are sometimes hardly met by standard, general-purpose modeling languages. The need to express domain features, as well as to express them using paradigms familiar to domain experts, calls for domain specific languages.

We propose a methodology to develop 3D graphical user interfaces for monitoring and controlling complex control systems. Instead of developing or specifying the interface directly, an automated prototype is generated from knowledge about the system under control. The methodology is comprised of a domain specific language for modeling control systems, and integrates a formal framework allowing model checking and prototyping. In the following sections we will describe the domain and goals of this project. A case study will be introduced to serve as a guide example. Section 2 will discuss the methodology from an abstract point of view; section 3 will give details on the technologies of the framework implementing the methodology. A related work section, conclusions and a future work overview will wrap up the article.

[5] This project is funded by the Hasler foundation of Switzerland

1.1 Domain definition

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

A complex control system will generally have a composite structure, in which each object can be grouped with others; composite objects can be, in their turn, components (or "children") of larger objects, forming a hierarchical tree in which the root represents the whole system and the leaves are its most elementary devices. Typically this grouping will reflect a physical container-contained composition (e.g. an engine contains several cylinders), but it could reflect other kinds of relations, such as functional or logic. 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 its subobjects.

Operators can access the system at different levels of granularity, and with possibly different types of views and levels of control, according to several factors (their profile, the conditions of the system, the current task being executed).

1.2 Requirements

The main goal of this project is defining a methodology that allows easy prototyping of a graphical user interface (GUI) for such systems. The prototyping has to be done by users who have a knowledge of the system under control; they should not necessarily have any deep knowledge of programming or GUI design. In this, our approach is different from several others which try to be general by focusing on GUI specification formalisms (see the Related Work section). This work proposes to model the system under control instead of the GUI. On one hand, this makes the methodology less general and only applicable to the domain of control systems (and possibly similar domains). But on the other hand, the methodology becomes accessible by people who don't have a specific GUI development know-how, and allows rapid prototyping by reusing existing information.

The requirements are the following:

- a system expert must be able to specify the knowledge of the system under control
- it must be possible to generate an executable prototype of the GUI from the specification
- it must be feasible to verify properties and to validate the specification
- it must be possible to classify users into profiles
- it must be possible to define tasks, which may be available only to some user profiles
- tools must be in place for accomplishing the previous requirements in a coordinated workflow

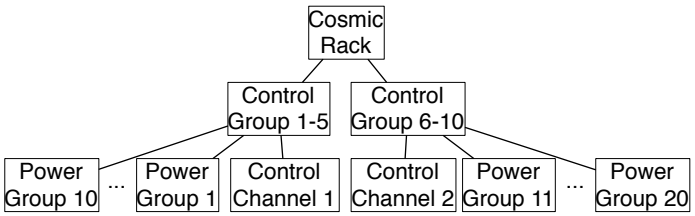

Fig. 1. Cosmic Rack hierarchy

1.3 Case study

As a reference for the examples in this article, we will use a case study from the European Laboratory for Particle Physics (CERN) in Geneva, published in [1].

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]. We worked on an early prototype of the Silicon Strip tracker, called the Cosmic Rack. This is equivalent to a section of the full tracker, mantaining the same hierarchical complexity, but with a reduced total number of components. The Cosmic Rack has been used to test the hardware and software of the full tracker. The hierarchical structure of the Cosmic Rack is shown in Fig. 1.

There are four types of components: *Partitions*, *Control Groups*, *Power Groups* and *Control Channels*. There is only one Partition, the *Cosmic Rack* object. There are two Control Groups; twenty Power Groups (ten per Control Group); and two Control Channels (one per Control Group).

Each component is characterized by a finite state machine (FSM). They are represented in Figg. 2 (for Partitions and Control Groups, which have the same FSM), 3 (for Power Groups) and 4 (for Control Channels).

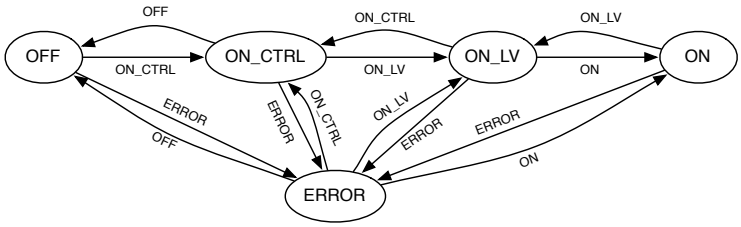

Fig. 2. Partitions and Control Groups FSM

The shape of every component, and its position in space, is defined by the Cosmic Rack mechanical project. This information is stored in a database.

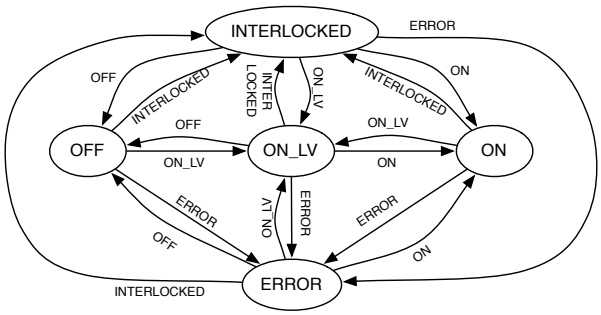

Fig. 3. Power Groups FSM

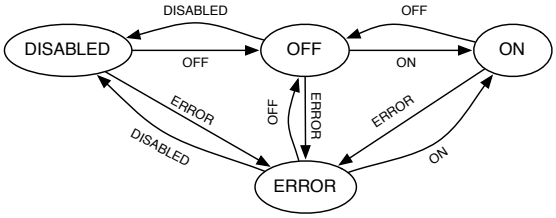

Fig. 4. Control Channels FSM

Each component can receive commands. They trigger transitions in the FSM. A command will generally trigger a transition having the same name as the command. There are some transitions, however, which are not triggered by commands, but rather by internal system events. It is the case for all transitions going to the "ERROR" state of all FSMs (triggered by property values out of range) and those going to the "INTERLOCKED" state of the Power Groups FSM (triggered by a hardwired security mechanism). In addition to this, Power Groups have a *Clear* command which triggers one of the transitions leaving the "INTERLOCKED" state (chosen according to current system values).

States are propagated up through the component hierarchy according to rule sets. Table 1 shows the rule set for propagating Control Groups (CG) states to the Partition. Each row is a rule. All non-empty conditions must be met for a rule to be applied. The table is a slight simplification of reality - there exist "mixed" states, for example when going from OFF to ON_CTRL, where only some of the Control Groups have already switched state. These mixed states, however, are normally ignored as they are not part of the "ideal" behaviour of the machine. Control Groups on their turn also have a similar table of rules for propagating the states of their children components, Power Groups and Control Channels.

Power Groups have a *temperature* property. On this property, an alarm is defined with temperature thresholds defining value intervals. Each interval corresponds to a diagnostic on the temperature (normal, warning, alert and severe).

Partition State	CG in OFF	CG in ON_CTRL	CG in ON_LV	CG in ON	CG in ERROR
OFF	ALL				
ON_CTRL		ALL			
ON_LV			ALL		
ON				ALL	
ERROR					ALL

Table 1. State propagation rules for the Partition. CG stands for Control Groups.

This is needed to detect temperature anomalies and take action before a hardware safety mechanism (based on PLCs) intervenes to cut power to components in order to preserve them (which brings a Power Group to the INTERLOCKED state).

There are command sequences which constitute the normal operation of the Cosmic Rack. These are turning on the system, turning off the system, clearing errors and clearing interlock events.

Turning on the system consists in turning on the Control Channels, then turning on the Power Groups.

Turning on the Control Channels means enabling (DISABLED → OFF) and then turning on (OFF → ON) the Control Channels (in any order).

Turning on the Power Groups means turning on low voltage (OFF → ON_LV) and then high voltage (ON_LV → ON) of Power Groups (in any order).

Turning off the system consists in the inverse sequence as turning on the system.

Clearing an error consists in sending commands to a Control Channel or Power Group in ERROR state, according to the situation at hand (most of the times, this will mean trying to power down a component and power it up again with a turn off / turn on sequence on the concerned branch).

Clearing an interlock state consists in sending a *Clear* command to an interlocked Power Group.

2 Methodology

The engineering process for prototyping a GUI for a control system is illustrated in Figure 5. There are four steps in this methodology. In the first step, knowledge about the control system is gathered. This is usually present under the form of a collection of more or less formal documents. In the case of complex systems, it is often the case that many of the aspects of the system are modeled in electronic form for engineering purposes. These models can have various levels of reusability depending on the format they are in. Knowledge about the system is essential because the composition of the system, its inputs and outputs, and its behaviour in terms of state evolution are key information for automated GUI prototyping. In the second step, this information is expressed using a domain specific language. This language models the domain of the information gathered

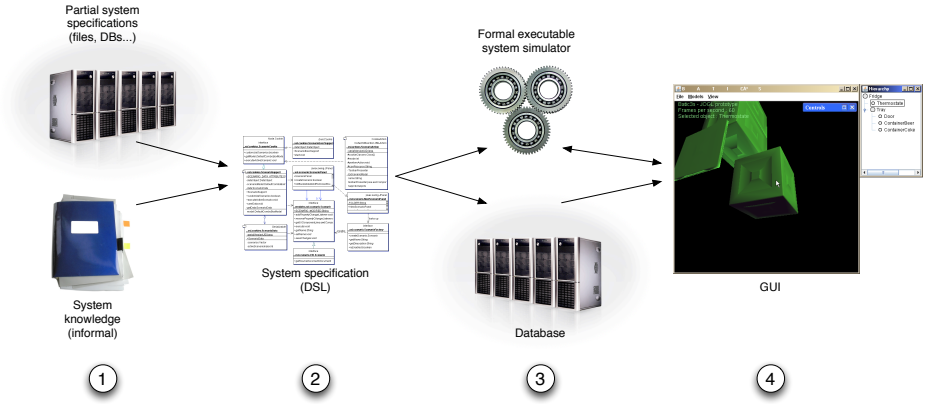

Fig. 5. The methodology process

in step one. This domain model is comprised of several models: one, the *system model*, describes the structure and internal behaviour of the system; the others, namely the *visual model*, the *user model* and the *task model*, describe rather the geometrical and interactive aspects of the system. An abstract overview of the models is shown in Figure 6. The various models are not completely separate, but are linked by several relationships, abstracted in Figure 6 by lines.

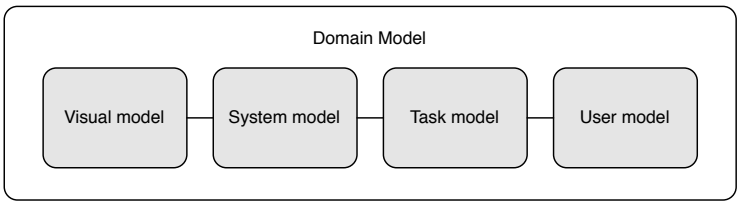

Fig. 6. Packages of the specification model. Arcs are abstractions of existing relationships among classes in the packages.

The third step of Figure 5 sees the generation of deliverables from this specification: a database containing data used for GUI generation (the visual, user and task model and part of the system model), and an executable system simulator. This is done by automated tools.

The fourth and final step is the dynamic generation of a GUI prototype built from the database data, which interacts with the system simulator.

Steps 2-4 are those tackled by our model-based approach. We will now describe the features of the domain specific model, then the simulator generation and GUI prototyping activity.

2.1 The domain model

We will now describe the sub-models of Figure 6.

The System model contains useful abstractions to describe the structure of the system and its behaviour. It includes the following concepts:

- Objects in the system
- Types, defining sets of similar objects
- A hierarchical composition relationship between objects
- Behaviour of objects in terms of states and transitions
- State dependency between components
- Properties of objects
- Commands and events of objects

Objects are identified by a name and represent components in the system.

Types are where we define all features which are not specific to individual instances of objects. Typing objects enables quick definition of properties common to a large number of objects (a typical situation in control systems which are highly repetitive). A type is identified by a name.

The hierarchical composition is modeled as a tree of objects. Each object can have one parent and/or several children. The hierarchical relationship can semantically express either physical containment or logical groupings (e.g. electrical connection, common cooling pipes...).

Behaviour of objects is modeled by finite state machines (FSMs). These are well-suited for control systems as they express expected behaviour in a clear way and are a standard in the control systems domain.

State dependency is expressed with conditional rules. Conditions can be of type "if at least one of the children (or - if all of the children) of an object is in state x, then go to state y".

Properties are identified by names and data types. Their possible values can be divided in intervals corresponding to four diagnostic levels: normal, warning, alert or severe.

Commands, as control systems are asynchronous by nature, are defined by their name and parameters only (i.e. no return value). For the sake of simulation, the possibility of defining a simplified behaviour for a command is provided (e.g. it is possible to specify that a command changes a property of an object).

Events are defined by their name and parameters. An event can be triggered by state transitions (also in children objects), command, property changes. An event can also trigger state transitions, commands and other events.

The behaviour of objects, their commands, events and properties are not directly associated in the model to individual objects, but rather to their types. This is because they are common features of all objects of a given type (e.g. a model of power supply). Individual objects instantiate types, inheriting all of the above features without having to specify them for each object. Figure 7 shows an abstract overview of the relationships among concepts in the system model.

Each arc in the figure sums up one or more relationships existing among the concepts. Individual relationships will be detailed in Section 3, while discussing the language metamodel.

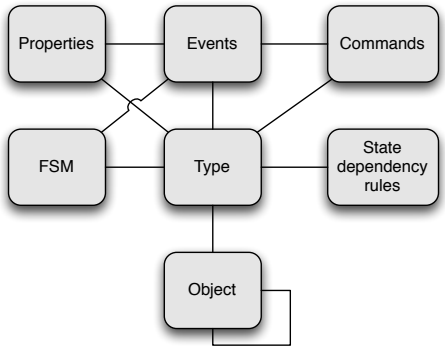

Fig. 7. Concepts in the system model and their relationships

The Visual model describes all aspects of the system which are visually relevant. This includes their geometrical space and their position in space.

Geometrical shape is defined for each type (and not per-object, for the same reasons mentioned earlier). The geometry is expressed by association to a geometry file URL. This is a file in the *Object* format[3], a commonly used standard for 3D object description. The language also includes pre-defined common primitive shapes (box, sphere, cylinder...) that are mapped to pre-made object files. Geometrical shapes can be modified by scaling attributes. For this reason a single shape can be used to model different objects. For example, using the cube primitive, one can model different cuboids by modifying its scale along one or more axis.

Position in space can be defined in different ways. The simplest is expressing translation and rotation for each object. This requires defining coordinates for each object. Components of complex systems, however, are sometimes positioned according to repetitive patterns (arrays, circles...). Thus, a more efficient choice could be positioning an object by specifying its relationship with other objects. Relationships of type "x is parallel to y", or "x is concentric to y" can be repeatedly applied to rapidly express whole sets of coordinates in such cases.

Figure 8 shows an abstract overview of the concepts of the visual model and their relationships. Individual relationships will be detailed in Section 3.

The User model, as a general definition, is a knowledge source that contains a set of beliefs about an individual on various aspects, and these beliefs can be decoupled from the rest of the system[4].

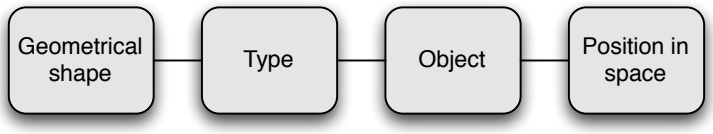

Fig. 8. Concepts in the visual model and their relationships (`Type` and `Object` are those from the System model)

We used the *Generic Ontology based User Model* (GenOUM) [5, 6] as a user model. This is a general-purpose user model based on an onlology including information about users' personality and knowledge. The GenOUM ontology is quite rich and goes beyond our needs. We are using a subset of it, represented in Fig. 9. The full GenOUM ontology is presented in [5]. We model what profile users have (*Behaviour*) and what is their level of knowledge for tasks to accomplish. The object of a user's knowledge in the ontology is generic (the *Thing* concept from OWL); we will replace *Thing* with the *Task* concept. The task model is described in section 2.1.

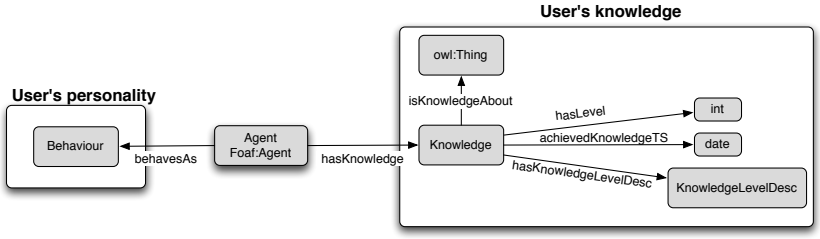

Fig. 9. GenOUM concepts and properties

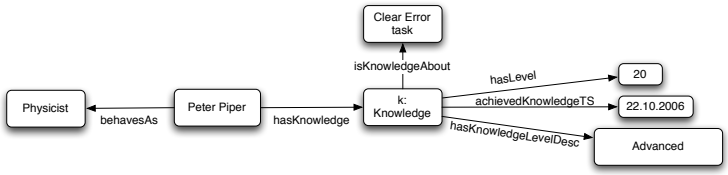

Fig. 10. Example of User's knowledge level

As an example referring to our Cosmic Rack case study, Figure 10 states that user Peter Piper has a physicist profile. He knows how to perform the "Clear error" task, his knowledge level about this task is 20, an advanced knowledge

level, and he acquired this knowledge on October 22, 2006. The interpretation of the knowledge level '20' depends on the domain and is not defined *a priori* by GenOUM.

The Task model: a task defines how the user can reach a goal in a specific application domain. The goal is a desired modification of the state of a system or a query to it [7]. Tasks must be seen as structured entities. As they represent the complex interaction between user and GUI, we should be able to deal with more than just atomic or linear tasks. Consequently, we chose to base our task model on the ConcurTaskTree formalism[7] (CTT). In CTT, a task is identified by a name, a type and an ordered list of subtasks, in a hierarchical composition structured like a tree. Temporal relationships between subtasks are defined.

There are four task types: *abstract*, *user*, *application* and *interaction*. An *abstract* task is generally a complex task we can define in terms of its subtasks. A *user* task is something performed by the user outside the interaction with the system (e.g. deciding or reading something). An *application* task is something completely executed by the system (e.g. cashing a coin inserted in a drink vending machine). Finally, an *interaction* task is performed by the user interacting with the system (e.g. clicking a button).

Concurrency relationships between tasks are defined by a number of process algebra operators (called *temporal operators* in [7]). A non-exhaustive list of these taken from [7] follows as an example:

- `T1 ||| T2` is *interleaving*: the actions of the two tasks can be performed in any order
- `T1 [] T2` is *choice*: T1 or T2 can be performed
- `T1 >> T2` is *enabling*: when the first task is terminated then the second is activated
- `T1*` is *iteration*: the task is iterative

CTT has a visual syntax, representing a task as a tree (where the root is the highest abstraction of the task, and the leaves are the most elementary actions) and is supported by editing tools and libraries. For the four task types mentioned, the symbols in Figures 11 are used.

Fig. 11. CTT task types: abstract, user, application and interaction

An example of CTT applied to our Cosmic Rack case study is shown in Figure 12.

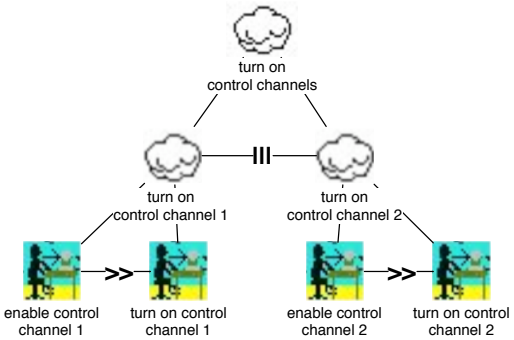

Fig. 12. CTT for the *turn on control channels* task

Through the association of task and user models, we add to the CTT model the possibility to specify for which user profiles (the *behaviour* in GenOUM) each task is available. This will be clearer later, when we give the detailed implementation of the task model.

2.2 System simulator generation

The system simulator is built by giving executable semantics to the syntactic specification of the control system. Through model transformation techniques (detailed in section 3), a formal concurrent model is obtained from the specification, as shown in Figure 5. This implements, notably, the instantiation of objects, the execution of the FSMs of the objects, the command input model, and the event model.

This model has executable concurrent and transactional semantics. Commands and events can be processed in a parallel and/or non-deterministic way, while the execution of command-event chains supports full transactionality and roll-back (to model command failures). The concurrent semantics greatly help in simulating a system which is, by nature, concurrent (several events can happen independently together in a control system). Transactionality, on one hand, might not be 100% corresponding to the actual system behaviour. In particular, it does not model system or communication failures. However, these are issues which are not specifically related to the GUI we are validating, and it is actually useful to abstract them by assuming that the system is always responding according to the specification.

2.3 GUI prototype

The GUI prototype engine is a software framework capable of loading the system specification from the database, and presenting the user with an interface allowing interaction with the system, as shown in Figure 5. A driver in this framework instantiates and runs the system simulator, so that interaction with the actual

system can be emulated. Commands and events are transmitted to and from the system simulator so that the users can evaluate the interface.

The task model information is used to know what tasks are available for a user when an object is selected. Thanks to the executable nature of the CTT formalism, it is possible to have the GUI automate a task, with the user only initiating it (at least in the case where only enabling operators are used). Alternatively, step-by-step cues can be shown to the user to perform the task (wizard-style). The user model is used as an authorization model for determination of the available tasks.

For the kind of systems we are modeling (geometrically complex, where error detection and diagnostic is important), we need an adaptation mechanism able to highlight components which are having errors. Difficulties in seeing a faulty object can come from it being out of the current view, and/or being hidden by other objetcs.

However, a low-level definition of the GUI's adaptive behaviour in the language would defeat one of the goals of the project, which is not to require a deep knowledge of HCI techniques by the user.

To address the problem, the GUI prototype engine implements a rule-based adaptation system. An *adaptation rule* is a concept which depends on user profile and current task, and that is triggered by an error event. The rule defines an adaptation *method*, which says what type of behaviour the GUI must adapt to react to the event. A set of rules has been defined to address common problematic situations which prevent object visibility. Strategies to solve these situations are centering the camera on an object, moving objects which block the view out of the way, or making them transparent. When an error event is produced, the rendering engine applies these strategies. If the object is not in the field of view, the camera moves to include it. If it is impossible to get an unobstructed view of the object, other objects in the line of sight are either made transparent (if they are not in error themselves) or moved aside (if they are).

2.4 Validation

Besides using the prototype as a basis for a production-level interface, the main interest of producing the GUI prototype and system simulator in this model-based way is the possibility of performing validation and verification techniques and refining the model accordingly.

Validation is an activity that checks if a model adheres to the original requirements (answering the question: *am I building the right model?*). From the point of view of a control system GUI, validation allows knowing if the GUI lets a user view the status of the system and interact with it in the expected way.

Many aspects can be validated in a GUI. Relevant ones are presentation, interaction and environmental ones[8], such as navigation, interaction, appropriate user profiling, correct error detection. But one of the most interesting aspects to validate are task models. For example, if a given system status is supposed to be corrected via a task, a few things we want to know are: if the user can view all of the information needed for executing the task (like viewing all objects involved);

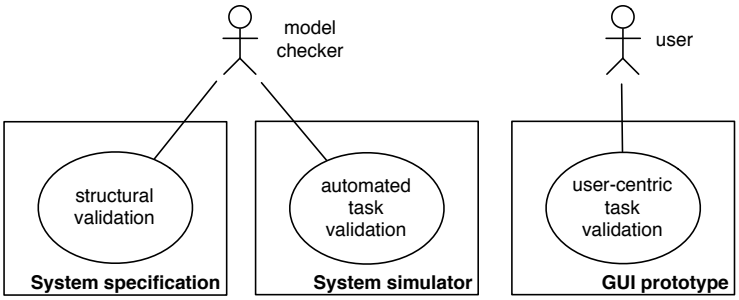

Fig. 13. Validation activities with their actors and models

if in all relevant cases the task is available; and in which cases the task may not be adapted to the circumstances. This type of validation can be led in two ways: first, an automated way, in which the task model is used to automatically run task commands in the GUI, and blocks or inconsistencies are detected; and second, a user-centric way, in which users are presented with a set of situations needing them to perform a task, and they evaluate manually the ease (or possibility) to perform it. In both cases, what we discover are misunderstandings of the requirements for the model. The result of this activity is an input for refining and improving the task model.

Another interesting aspect to validate is the structure of the system, as we need to be sure we are modeling the right system (and thus, the right GUI). Properties on the structure of the system can be specified using a constraint language in the model, and these constraints can be checked at design time, in a static way. We will be able to check if, for example, a certain type of object always has children of a certain other type, or if all interaction tasks are associated to a command. The actors in the discussed validation activities are shown in Figure 13 together with the involved products of the methodology.

2.5 Verification

Verification is an activity allowing to know if a model has been correctly made (answering the question: *am I doing the model right?*). In our domain, this means for example if all transitions in an object's FSM have the correct source and target state, or if an object's state change is correctly reflected in its parent object's state.

There are two types of interesting verification in this methodology. The first focuses on the model itself and tries to make sure that the simulator we run against is respecting the actual system properties. This type of verification can be performed on the system simulator (which has executable semantics and supports state space generation and exploration), by specifying properties in a suitable formalism, like temporal logic. Model checking tools can evaluate these properties on the state space of the simulator and confirm if the model satisfies them. We can also use testing: feeding the simulator a known input and detecting

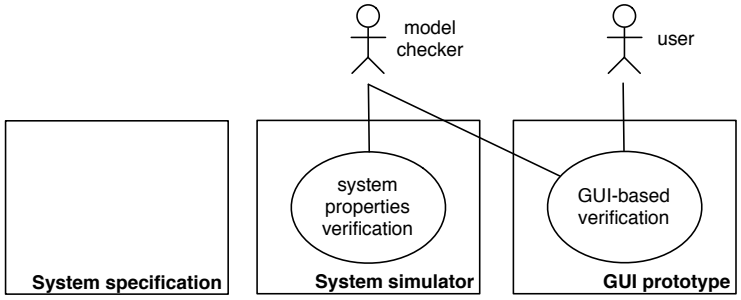

Fig. 14. Verification activities with their actors and models

if the output matches expected values. The second type of verification is on the GUI-related aspects, like task and user models, geometry, and adaptation rules. It checks, for example, if the geometry respects the system structure, or if a rule will always be triggered when its conditions are met. This type of verification can be done via property specification and model checking or by manual testing. What we discover are errors in the model specification that have to be corrected.

In both cases (refinement and correction), a model-based methodology is useful because one only needs to modify the original model, and all the code generation and prototyping is redone automatically. The actors in the discussed verification activities are shown in Figure 14 together with the involved products of the methodology.

3 Framework

The methodology described in the previous section has been implemented by integrating a set of tools. We preferred choosing well-known and open tools and frameworks, filling the gaps with ad hoc-developed tools when necessary.

We will now describe the various phases of the methodology with concrete references to the technologies and methods we propose.

3.1 The specification language

One of the purposes of the methodology we propose is to introduce a comprehensive domain specific language for the domain of control systems that is based on useful abstractions for domain experts. The modularity of our domain model led us to design a DSL called COntrol system SPEcification Language (Cospel) that is made of different parts, or *packages*, modeling different aspects of the system. We used the Eclipse Modeling Framework[9] (EMF) to specify the abstract syntax of Cospel by defining its metamodel. The semantics have been given by transformation to a different formalism, which will be described in section 3.2.

Cospel specifications can be created via an editor, generated automatically by EMF from the abstract syntax. The concrete syntax provided by this editor

is a tree-like visualization of the specification, where each node is a syntactic element. The tree is structured according to the aggregations in the abstract syntax. Properties and associations other than aggregations can be specified by editing properties of tree nodes. Specifications are serialized in a special XML format. The editor runs as an Eclipse application.

Based on extensions of related work[10], Cospel is composed of the following packages (following the order in which models have been introduced in section 2).

The Cospel core package is part of the System model. It defines the hierarchical structure of the control system. Figure 15 shows the metamodel of this package. The `Specification` element is the top level element for any Cospel specification. It serves as the container for all specification elements. The `Type` element serves as a template for similar objects: all features which are common to a number of similar objects can be modeled only once in their type. This choice is motivated by the highly repetitive structure of control systems, and is supported by the common practice of using such a template system to reduce the workload of specification in the domain. The other key concept here is the `Object`. It represents an individual component, and it models the hierarchical structure of the control system (an object can be the child of another object). All other features of the system are modeled in other packages and associated with either the type (for features common to many objects) or the object (for object-specific features).

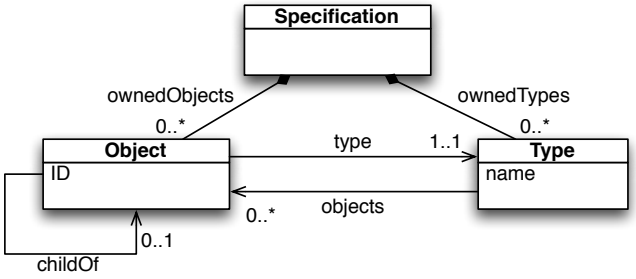

Fig. 15. Cospel core metamodel

With respect to our Cosmic Rack example of Figure 1, we model here four types: `PARTITION`, `CTRL-GRP`, `CTRL-CHN` and `POWER-GRP`. Individual objects are then defined (and associated to their type): `CosmicRack`, `CG1-5`, `CG6-10`, `Ctrl-Channel-1`, `Ctrl-Channel-2`, and all Power Groups from 1 to 20 (numbered according to a layer and index convention, e.g. `PG-Layer-4-Rod-2` is number 8).

The Finite State Machine (FSM) package is part of the System model. The behaviour of objects in terms of states and transitions is modeled via a simple FSM model. This is supported by FSMs being well known and understood in the domain. An `FSM`, comprised of `States` and `Transitions`, is associated to a `Type` (from the core package). This means that all objects of that type will have that FSM. Figure 16 shows the FSM package metamodel.

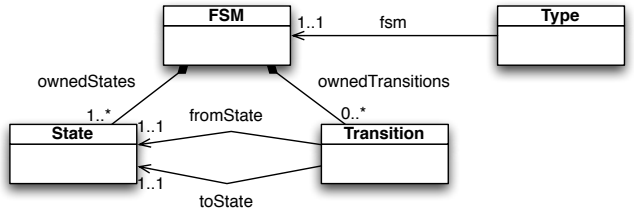

Fig. 16. FSM package metamodel

Using this metamodel we can, for example, model the FSM of a Control Channel (from Figure 4). We create four states: `DISABLED`, `OFF`, `ON` and `ERROR`. Also, we define a transition for each arc in the FSM. These states and transitions are grouped in an FSM called `FSMControlChannel` which is associated to the `CTRL-CHN` type.

The State dependency rules package is part of the System model. Recalling what was said in section 2, we want to be able to define rules in the form *"if all of the children of this object are in state x, set this object to state y"*. This is achieved by associating a state composition rule (`StateCompRule`) to a type. All objects of that type will implement this rule. The rule is associated to a state (representing the children objects' state) via a *childrenState* relationship. It is also associated to a resulting state via a *resultingState* relationship. `StateCompRule` also has an `orAndType` attribute. This defines if the rule is triggered when *all* children are in a given state, or when *at least one* child is. Figure 17 shows the metamodel of this package.

Fig. 17. State composition rules package metamodel

For the Cosmic Rack example, referring to Table 1, we declare a `PartitionErrorDependency` rule, with the `orAndType` attribute set to `And`. This

rule is associated with the `PARTITION` type. Through the *childrenState* relationship, it is associated to the Control Groups `ERROR` state. Through the *resultingState* relationship, it is associated to the Partition's `ERROR` state. Other rules implementing the table can be defined in the same way.

The Property package is part of the System model. It defines the *Property* class, associated to a type. All objects of that type will have the property. The property has a data type. It also has attributes defining value intervals associated to diagnostic alerts (normal, warning, alert and severe). For the Power Groups in the Cosmic Rack example, we define the temperature property, an integer value. For each of the four diagnostic values, we define upper and lower temperature limits.

The Command and Event package is part of the System model. The metamodel in Figure 18 describes events.

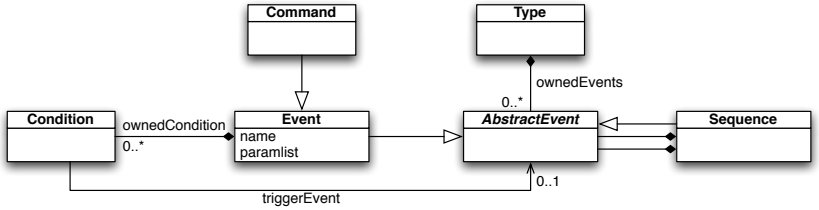

Fig. 18. Event package metamodel

An `AbstractEvent` is associated to a `Type`, meaning that all objects of that type will have it. It can be either a simple `Event`, or, recursively, a `Sequence` of events. A simple event has a name and a list of parameters. A `Command` is a specialization of an event. Based on the satisfaction of a `Condition`, an event can be executed, and can optionally trigger another event. The event can also be associated (not shown) to a transition of the type's FSM; this means when the event is executed, the transition is triggered. Conditions are characterized by expressions, evaluated on properties and/or event parameters, and include the definition of pre-postcondition expressions. Multiple conditions can be used to axiomatize an event.

The Control Channels in our Cosmic Rack case study have an `Enable` command. This command has no parameters, thus its condition does not state any particular precondition for its execution. However, `Enable` is associated to the `OFF` transition going from state `DISABLED` to state `OFF` (see Figure 4). This implies that the command is only executable if the control channel is in state `DISABLED` (otherwise the transition would not be fireable).

The Geometry package corresponds to the visual model. A `Geometry` class (abstract, that generalizes several classes like `Box`, `Cylinder`, `GeomFile`...) is associated to a `Type`, defining the shape of all objects of that type. A `Scale` class is also associated to a `Type`, allowing the reuse of the same geometry at different scales for different types (e.g. one might have two types of screws, identical but for the length). Classes defining position in space and rotation are directly associated to the object, placing the object in space. For the position in space various possibilities exist: giving absolute coordinates, or placing the object with relationships to other objects (parallel surfaces, distance...). The metamodel of this package is in Figure 19, with a simplification on the *relationship* class (which generalizes a number of possible relationships). Note that having a geometry/position is optional; this is because we could make models in which there are some objects which are only *logical* objects. They do not corresponding to a physical object, but are only used to group other objects for diagnostic purposes.

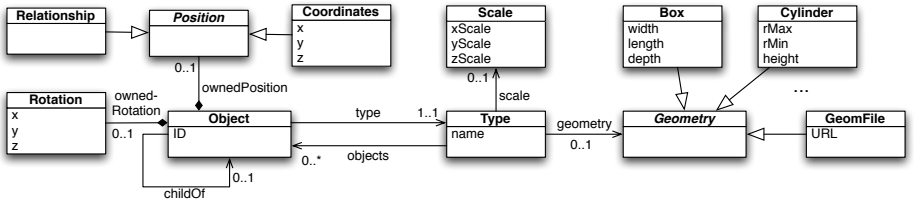

Fig. 19. Geometry package metamodel

In the Cosmic Rack, all components are cuboids (although with different sizes, proportions and orientations). To model their geometry, we define a `Box` with unitary dimensions and associate it to all types. Then we associate a different `Scale` to each of the four types, giving the size in the three axes of the box. Also, all Power Groups with a name ending in "Rod_1" have a rotation on the X axis, so we define a `Rotation` and apply it to all those objects. Thus, to characterize all shapes, we only define one geometry, one rotation and four scales. We then position each object in space according to the mechanical drawings of the Cosmic Rack.

The Tasks package corresponds to the Task model. Building the metamodel for the CTT formalism, we chose to use binary task trees to avoid the problem of defining a priority for the temporal operators, as suggested in [7]. The resulting metamodel is shown in Figure 20. An individual `Task` can be associated with a `Type`; this means that the task will be available for every object of that type. A task can also be associated with a `Command` from the Event package, which means that performing the task involves sending that command to the object associated with the task. Finally, a task is associated to a `Profile`, a concept described in

the user model in the following paragraph, meaning that only certain profiles are allowed to perform that task. Tasks are related in a tree structure by `Operator`s (here we show only three of them).

Modeling the *turn on control channels* task for Control Channel (Figure 12), we create a task for each CTT node (*turn on control channels*, *turn on control channel 1*...). Each task is associated to the `CTRL-CHN` type. The leaf nodes are associated to the `Enable` and `TurnOn` commands of the Control Channel. We relate the tasks using the enabling (`>>`) and interleaving (`|||`) operators.

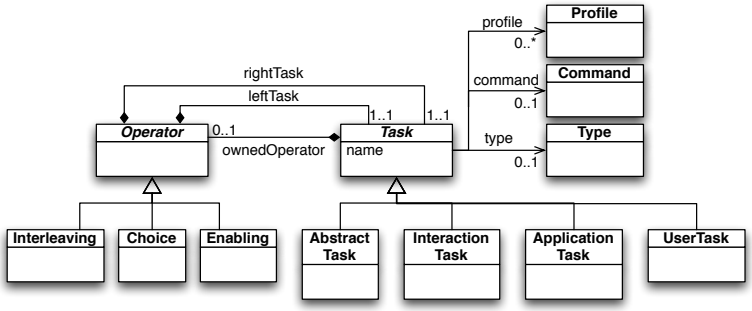

Fig. 20. Task package metamodel

The Users Package corresponds to the User model. The metamodel corresponds to what has already been shown in Figure 9, with the *Thing* class replaced by the `Task` class from the task package. We also renamed *Behaviour* to *Profile* for clarity. The metamodel is shown in Figure 21.

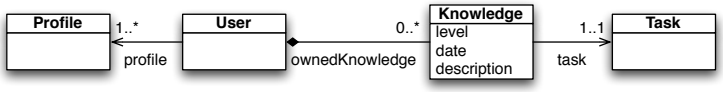

Fig. 21. User package metamodel

We already showed an example for the Cosmic Rack in Figure 10: we create a `Profile` called `Physicist`, and a `User` named `Peter Piper` associated to this profile. We associate Peter to the task model for the `Clear Error` task through an instance of `Knowledge`, which is characterized by a `level` of 20, and a `date` of 22.10.2006.

3.2 Transformation

To give semantics to a Cospel model, we use model transformation techniques. Instead of defining a semantic for each element of the Cospel language, our approach establishes mapping rules between the Cospel metamodel and the CO-OPN[11] metamodel. CO-OPN (Concurrent Object-Oriented Petri Nets) is an object oriented modeling language based on algebraic Petri nets, allowing the execution of specifications and providing tools for simulation, verification and test generation. CO-OPN support concurrency and transactionality, and there are tools to generate executable Java code from a CO-OPN model. Since the semantics of CO-OPN are already defined (in formal terms), we obtain the Cospel semantics as a result of the transformation.

In the context of the BATIC^{3}S project we chose to use the Atlas transformation language (ATL) framework[12], a declarative, rule-based language and framework for specifying mapping rules in language transformations. ATL is particularly well suited for its declarative style and its modularity. The ATL framework is very usable, and runs as an Eclipse plugin, another advantage for our methodology which is mainly based on Eclipse-related tools.

We followed a modular approach, identifying the different packages of the Cospel language (e.g. tasks, users, object hierarchy...) as sources for the transformation; for each module, we gave a transformation pattern with ATL rules. The patterns have then been composed, with syntactic and semantic composition techniques, to obtain a set of transformation rules able to transform the whole Cospel framework.

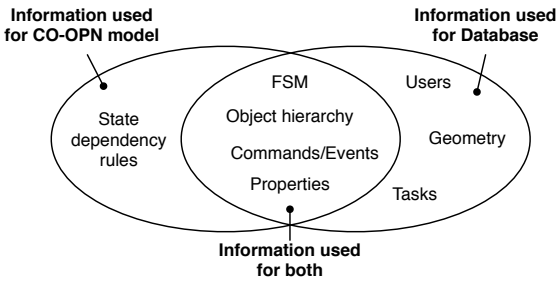

Fig. 22. The CO-OPN model and the database are built by transformation of Cospel packages; some packages are used to produce both models, while others only contribute to one of them

As we said before, the methodology's artifacts are two: a system simulator and a GUI. To produce them, the ATL transformation creates two models from the Cospel specification. The first one is the CO-OPN model which has just been described. Since the system simulation is purely behavioural, no visualization-related information is used for building the CO-OPN model. The second model created by the ATL transformation is a database, which has to be used as a

data source for creating the dynamic 3D scene in the GUI. Information including object hierarchy, geometrical data, FSMs (for representing states), tasks, properties and commands/events (to create interaction methods) are stored in this database. Figure 22 shows which Cospel packages contribute to which model.

An overview of the transformation process is given in Figure 23, showing what the players in the transformation of the Cospel model to a CO-OPN+Database model are. It is shown how the individual language packages and their transformation patterns are composed into a global metamodel and a global transformation. A detailed description of this composition and transformation process is in [10]

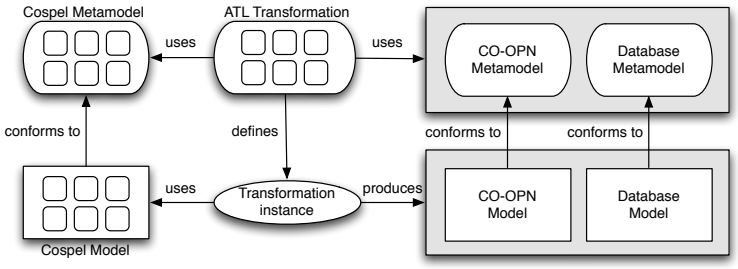

Fig. 23. Transformation overview. Smaller rectangles inside Cospel metamodel and model and inside the ATL transformation represent individual language packages and their individual transformation patterns

Without getting too deep in the details of the CO-OPN specification resulting from the Cospel transformation, we will now describe its structure. We used the object-oriented features of CO-OPN to create the hierarchical structure, by using Cospel types as a template for instantiating CO-OPN objects. Commands are transformed to CO-OPN methods, and events to CO-OPN gates (a kind of parameterized event). Since CO-OPN objects contain an algebraic Petri net, Cospel object states and properties are translated to CO-OPN places, and behavioural rules (such as FSM transitions or dependency rules) are translated to axioms. Figure 24 shows the structure of a part of the Cosmic Rack hierarchy using CO-OPN's visual syntax. It shows how objects are instantiated inside imbricated *contexts* (a CO-OPN context is an encapsulation allowing coordination among object instances), how commands are routed (arrows) down the hierarchy until their destination object, and how events are routed up. The routes (called *synchronizations*) are decided based on conditions (e.g. the name of the destination object, or the parameters of the command/event), a part of which comes from the specification, and the rest are automatically generated in the transformation patterns.

Verification can be performed on the CO-OPN model by using decision diagrams. Tools are under development[13] for model checking CO-OPN. These tools implement decision diagrams model checking for algebraic Petri nets. This

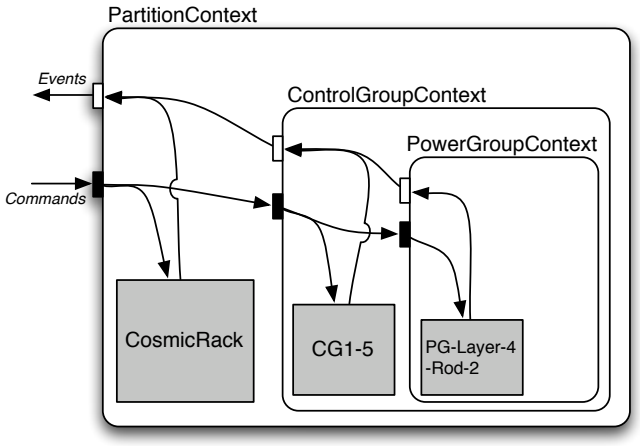

Fig. 24. Hierarchical structure of CO-OPN model

allows exploring a specification's state space to check whether properties are satisfied. Verification of temporal properties is thus possible (for example properties concerning dynamic behaviours, like tasks, events or interaction).

The CO-OPN specification is finally used by the CO-OPN tools to generate Java code. This code constitutes the formal executable system simulator mentioned in Figure 5. It serves as the executable basis for evaluating the GUI prototype.

3.3 GUI engine

The GUI engine which creates the GUI for the user to interact with is written in Java. It presents a 3D rendering of the system which allows spatial navigation and interaction with system components. Figure 25 shows the data flow of the GUI engine. Its source of information for rendering the scene are the database (DB) produced in the transformation phase, and the set of Object files containing the shapes of the system components. By loading these resources, a 3D scene is initialized and shown to the user. At the same time, the CO-OPN system simulator produced in the transformation is instantiated and initialized.

To minimize memory usage (a serious problem for systems with a large number of components), features common to sets of objects (i.e. those features associated in the specification language to types rather than to individual objects, like geometrical shape) are loaded only once and stored in instances of a `Object3DStructure` class. Individual objects are obtained by instantiating an `Object3D` class which applies individual features (e.g. position in space) to the associated `Object3DStructure` instances.

Objects are organized in a tree, according to the hierarchy in the specification. Each of them keeps an instance of FSM, and the current state of an object is

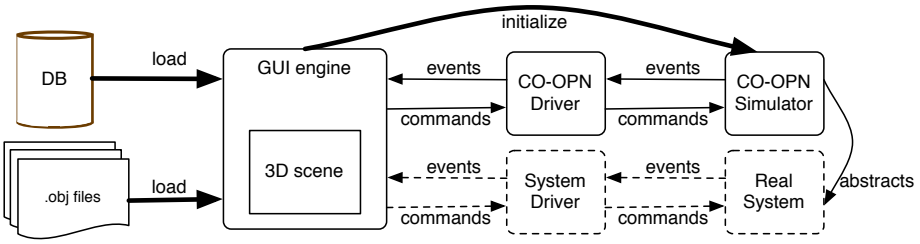

Fig. 25. GUI engine communication

represented in the scene by colors, according to common practices in the field (green=ok, yellow=warning, red=error, blue=powered, gray=off or unknown).

Upon selecting objects, users are shown with clickable controls corresponding to the commands defined for that object. These, when clicked, send a command message to the driver, which transmits it in the appropriate format to the CO-OPN simulator. Events coming from the simulator are also translated by the driver and can be interpreted by the GUI to update the scene (e.g. an object state change will modify its color, or trigger an error message).

After the development phase, when the prototype can be considered final, the CO-OPN driver can be substituted by a driver for the real system (see Figure 25), so that the interface can be evaluated in a real-life environment.

Rendering is done by using the JoGL API[14], a Java binding of the OpenGL libraries developed by Sun and Silicon Graphics. JoGL provides all necessary methods for 3D rendering and navigation. In addition to that, it has built-in support for stereoscopic visualization, which allowed us to experiment with how stereoscopic perception can affect interaction and navigation in a control system. We also used the FengGUI API[15], which allows drawing 2D components, like windows and buttons, in a 3D stereoscopic scene.

There are two navigation modes: spatial and hierarchical. Different controllers can be used for spatial navigation: keyboard, mouse (via click and drag gestures) or 3D controllers (e.g. six degrees of freedom knobs like 3D Connexion's SpaceNavigator [16]). Hierarchical navigation is done via a tree-like representation of the system.

The two navigation modes answer different types of tasks. When investigating a faulty component, the user may be interested in checking the nearby components' values to prevent further alarms, especially when the faults are of environmental nature (temperature, humidity). These components might well be "far" in terms of system hierarchy (i.e. belonging to a whole different branch of the hierarchy), but will be at a short distance in the 3D scene. The spatial navigation system provides easy access by moving the camera and directly clicking on them. On the other hand, if a user wants to quickly jump to another part of the system following an error, spatial navigation is not efficient as it requires several zooming, panning and rotating operations. In these cases, a few clicks on the tree representation provide a quick focus switch to another region of the sys-

tem. Also, the tree representation can help selecting objects which are difficult to pick (very small or hidden by other objects).

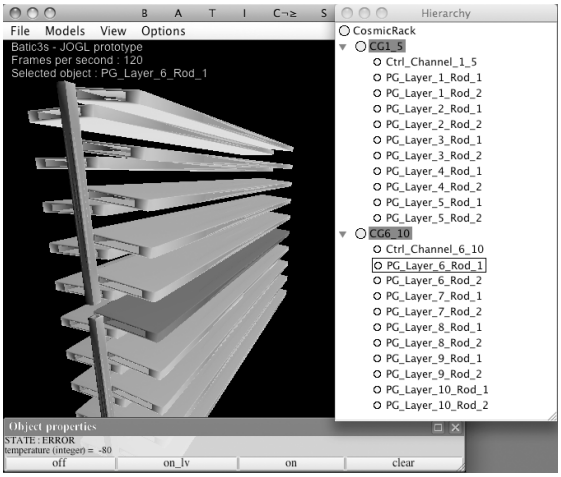

Fig. 26. GUI prototype screenshot. The Power Groups of the Cosmic Rack can be seen in the 3D scene

Figure 26 shows a screenshot of the GUI prototype. A 3D representation of the Cosmic Rack is shown, and Power Groups are visible. Objects containing Power Groups (i.e. the Control Groups and the Cosmic Rack external shell) have been made invisible, in order to show one of the Power Groups which is having an error (it can be seen in a different color). The 3D scene is built using the information about the system hierarchy, FSMs and geometry.

The user middle-clicked on the component, which brought up possible commands in the bottom panel. The temperature property is also shown here. The engine knows what commands and properties to show because they had been specified in the model.

The right panel marks the name of the currently selected component in the hierarchical tree view of the system. Since the model says that the current user has access to the *off*, *on_lv*, *on* and *clear* commands of the Power Group, all these buttons appear in the bottom panel. A user with monitoring responsibilities only would not see the buttons.

4 Related work

4.1 Control systems and GUI models

There are two main lines of research related to this work. The first is about specification languages for control systems. The second concerns techniques and languages for GUI generation.

Several models have been proposed in the domain of control systems to be used as a specification language. RSML[17] exploits the fact that control systems are reactive and evolve their state depending on conditions. It is based on a modified version of Statecharts. It defines a mapping to an underlying formal model based on Mealy automata, but presents the user with a set of more readable syntaxes. The Fujaba tool[18] treats decentralized control systems. It models systems using a UML class diagram syntax, to which users can add detail such as Statecharts or collaboration diagrams. It supports specification simulation and Java code generation, but no model-checking features (except for those which can be applied to Java, of course).

VEG[19] treats two-dimensional GUIs as reactive event driven systems *per se*. It proposes a scalable notion to specify, design, validate and verify a GUI. Specifications are modular, separated into control and data, and are layout-independent. The syntax supports event-based behaviours, parallel composition and runtime instantiation. Java code functions can be added to model complex behaviours. VEG offers model checking possibility in that specifications can be transformed to Promela and model-checked with SPIN[20]. A Java prototype of the GUI can be generated.

In the domain of GUI generation techniques, various approaches tackle specification and prototyping. Bastide et al. [21] use a formal object-oriented formalism (ICO) to describe interactive applications. The developer can specify the interface behaviour as an Object Petri net. The GUI presentation is specified using the commercial tool JBuilder. Verification and validation can be performed because of the formal semantics of ICO, as well as prototype generation. There is no simulation of the system under control, the specification process demands a knowledge of Petri nets, and the specification of the presentation is manual.

Calvary et al. [22] define a general, unifying reference framework for classifying multi-target interfaces. Four levels of abstraction are defined (task and concepts, abstract UI, concrete UI, final UI). This framework is instantiated for a few GUI generation approaches, including ARTStudio [23], a tool by the same authors to generate multi-target GUIs from tasks and data specification. Here also, the simulation of the system under control is not one of the goals. The focus is abstraction and flexibility, and the definition of a generic framework which can be used as a reference to compare or unify similar approaches.

UsiXML (several papers in [24]) is an abstract user interface description language, allowing specification of the interactive aspects of a GUI. It can be rendered on different platforms and as different types of GUIs thanks to its level of abstraction. This flexibility makes it suitable as an underlying format for storing the interface model, and its use is currently being evaluated by our project.

The AID project [25] has a rather similar scope to our work. One of the fundamental goals of AID is generating a GUI from an existing model of the system under control. The syntax and semantics of the system model (called *Domain Model*) are used to build an interaction model and a presentation model. The task model for the interface is part of the domain model, which also is the

case for our project. However, this approach is mainly focused on automatic interface design; it does not provide simulation of the system, nor verification and validation activities.

Finally, Vanderdonckt et al. developed the Envir3D tool[26], which can generate a virtual 3D GUI from an abstract model of the interaction elements, dialog and presentation. The underlying model, hidden from the developer, allows evaluation, model checking and verification of the GUI. The main focus of the work is evaluating usability with respect to standard guidelines. There is no simulation of the system under control. This project complements our work, in that we focus a lot on system semantics and not really on ergonomics of the GUI, while the Envir3D project does the opposite.

4.2 Task models

A very in-depth survey about task models has been done in [27]. It shows and discusses several task models from the 1960's to early 2000's, thus we will not rewrite it here in detail. All reviewed task models express goals, task hierarchies, temporal constraints and role specification in terms of tasks. When it comes to comparing models however, the following factors differentiate and motivate the choice CTT from other reviewed choices:

- Discipline of origin: some other reviewed models are based on disciplines aside from software engineering (i.e. psychology or cognitive science) and give too much relevance to behavioural or cognitive aspects for our context
- Formalization: semantically, CTT is based on process algebra, which is suitable for the domain and activities we treat, while other models are not based on an underlying formal system
- Context of use variations: most reviewed models do not support task variations based on context of use, while CTT supports conditional subtrees of tasks
- System response: some of the reviewed models do not distinguish any type of task from the technical system, while CTT does

4.3 Adaptation models

In AWE3D [28], the authors focus on proposing a general approach to build adaptive 3D Web sites, and illustrate a specific application of the approach to a 3D e-commerce case study. In SLIM-VRT [29], the authors present the design and implementation of an efficient strategy for adapting multimedia information associated to virtual environments in the context of e-learning applications. They propose a strategy for VRML scenes which consists in separating the multimedia information to be associated to the virtual scene from the 3D models themselves. In AMACONT [30], the authors present an approach to deploy the adaptive hypermedia architecture AMACONT [31] together with the component oriented 3D document model CONTIGRA [32] to achieve various types of 3D adaptation within Web pages. The authors present implicit rule-based media

adaptation observing users interactions.In [33, 34], the authors present a system that is able to deliver personalized learning content in 3D Educational Virtual Environments. The techniques extend those employed in the field of Adaptive Hypermedia by the well known AHA! system [35]. In [36], the authors have presented an ontology-based model which is well-suited to create adapted views of an urban planning project. Adaptation is based on two factors: the user's profile, which is used to select a viewpoint-ontology; and the user's current centre of interest, which corresponds to a theme in the ontology of the themes. The generation of the links in the interface is carried out according to generation rules that correspond to different linking semantics. In 3DSEAM [37, 38] , the authors present an open framework supporting rule-based adaptations of 3D scenes is used. The main role of the framework is to arrange the adaptation process following an adaptation strategy, materialized by rules that come with the scene request. The adaptation framework relies on the 3DAF (3D Annotation Framework) that handles the identification of objects matching the rule criterion, and external engines that either adapt individual objects or regions [39].

5 Conclusion and perspectives

We presented a methodology to prototype adaptive graphical user interfaces (GUI) for control systems. We introduced a DSL for the control systems domain, called Cospel, based on useful and understandable abstractions for domain experts. Transformation techniques and semantic mapping to a formal model allow for simulation, validation, verification and automatic GUI prototype generation. Our approach is based on the assumption that a GUI can be induced from the characteristics of the system to control.

Several perspectives will guide the future of this work:

Improving the language: the modular structure we used for metamodels and transformations allows for a relatively easy extension of Cospel. Features can be added, such as defining a type-based template system for the hierarchy or specifying the behaviour of commands in a more complex way, or again modeling the interaction between the GUI and the system.

Enriching user information: the user model we have defined is actually richer than what we currently use. By implementing metrics in the GUI engine, we could measure factors like user mood, learning style and cognitive style based on how the user interacts with the GUI. This information could be used for evaluation of the GUI prototype.

3D adaptation: apart from open issues of defining procedural versus declarative 3D adaptation, work is ongoing on defining ontology-based 3D adaptation. We are currently investigating a case study for building 3D adaptive GUIs based on ontologies for urban planning communication[36]. Urban planning is concerned with assembling and shaping the urban, local or municipal environment by deciding on the composition and configuration of geographical objects in a space-time continuum. The main characteristics of the ontology-based model are the semantic integration in a knowledge base of the urban knowledge coming

from various sources (geographical information systems databases, master plans, local plans); and the modelling of the centre of interest of an urban actor. These models can be then used to generate adapted GUIs to present the project's data and knowledge according to each actor's background and interests.

3D stereoscopy impact: we want to evaluate if, in this domain, stereoscopy helps navigation; if immersion is relevant to knowledge representation; if there are unexpected side effects of using a 3D interactive environment; and if there is an advantage in using haptic devices or multitouch interaction.

Introducing ergonomy and usability criteria: while our current work is mainly interested in the semantic and methodological aspects of GUI generation, one should not forget that usability and ergonomy of a GUI are capital factors in its success. Existing approaches for applying standard usability metrics to GUIs should be integrated in the prototyping process.

We are also interested in extending the methodology to similar but distinct domains. Apart from the case study mentioned in section 1.3, we did another study [40] on an interface for data acquisition at a high energy physics experiment at CERN, called ATLAS. Results of the case study are still under evaluation, especially for what concerns the role of 3D representation in this particular domain.

References

1. Risoldi, M., Masetti, L., Buchs, D., Barroca, B., Amaral, V.: A model-based methodology for control systems gui design prototyping. In: Proceedings of the PCAPAC 2008 conference, available online at `http://www.pcapacworkshop.org`. (2008)
2. Dierlamm, A., Dirkes, G.H., Fahrer, M., Frey, M., Hartmann, F., Masetti, L., Militaru, O., Shah, S.Y., Stringer, R., Tsirou, A.: The CMS tracker control system. Journal of Physics: Conference Series **119**(2) (2008) 022019 (9pp)
3. Burkardt, J.: Object file format specification. `http://people.scs.fsu.edu/~burkardt/txt/obj_format.txt` visited in 2008.
4. Kobsa, A., Wahlster, W., eds.: User models in dialog systems. Springer-Verlag New York, Inc., New York, NY, USA (1989)
5. Cretton, F., Calvé, A.L.: Working paper: Generic ontology based user modeling - GenOUM. Technical report, HES-SO Valais (2007)
6. Cretton, F., Calvé, A.L.: Generic ontology based user model: GenOUM. Technical report, Université de Genève (June 2008) `http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=46`.
7. Paternò, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A diagrammatic notation for specifying task models. In: INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction, London, UK, UK, Chapman & Hall, Ltd. (1997) 362–369
8. Yip, S., Robson, D.: Graphical user interfaces validation: a problem analysis and a strategy to solution. In: System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on, IEEE Computer Society (1991)
9. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse series. Addison Wesley (2004)

10. Pedro, L., Risoldi, M., Buchs, D., Barroca, B., Amaral, V.: Developing domain specific modeling languages by metamodel semantic enrichment and composition: a case study. In: Submitted to the IEEE Software Special Issue on Domain-Specific Languages & Modeling. (2008)
11. Biberstein, O.: CO-OPN/2: An Object-Oriented Formalism for the Specification of Concurrent Systems. PhD thesis, University of Geneva (1997)
12. ATLAS Group: Atlas transformation language (2008) `http://www.eclipse.org/m2m/atl/`.
13. Hostettler, S.: Java decisions diagrams library. Technical report, Université de Genève (June 2008) `http://smv.unige.ch/tiki-list_file_gallery.php?galleryId=46`.
14. JoGL expert group: JSR 231: Java™binding for the OpenGL® API. `http://jcp.org/en/jsr/detail?id=231` visited in 2008.
15. FengGUI developer group: FengGUI: Java GUIs with OpenGL. `http://www.fenggui.org` visited in 2008.
16. 3DConnexion: Spacenavigator product web page. `http://www.3dconnexion.com/3dmouse/spacenavigator.php` visited in 2008.
17. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Transactions on Software Engineering **20**(9) (1994) 684–707
18. Kohler, H.J., Nickel, U., Niere, J., Zundorf, A.: Integrating UML diagrams for production control systems. icse **00** (2000) 241
19. Berstel, J., Reghizzi, S.C., Roussel, G., Pietro, P.S.: A scalable formal method for design and automatic checking of user interfaces. In: ICSE '01: Proceedings of the 23rd International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2001) 453–462
20. Holzmann, G.J.: The SPIN Model Checker. Addison Wesley (2003)
21. Bastide, R., Navarre, D., Palanque, P.A.: A tool-supported design framework for safety critical interactive systems. Interacting with Computers **15**(3) (2003) 309–328
22. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interacting with Computers **15**(3) (2003) 289–308
23. Calvary, G., Coutaz, J., Thevenin, D.: A unifying reference framework for the development of plastic user interfaces. In: EHCI. (2001) 173–192
24. UsiXML consortium: UsiXML - Home of the USer Interface eXtensible Markup Language. `http://www.usixml.org` Visited in 2008.
25. Penner, R.R., Steinmetz, E.S.: Implementation of automated interaction design with collaborative models. Interacting with Computers **15**(3) (2003) 367–385
26. Vanderdonckt, J., Chieu, C.K., Bouillon, L., Trevisan, D.: Model-based design, generation, and evaluation of virtual user interfaces. In: Web3D. (2004) 51–60
27. Limbourg, Q., Vanderdonckt, J.: Comparing task models for user interface design. In: The handbook of task analysis for human-computer interaction. (2004)
28. Chittaro, L., Ranon, R.: Dynamic generation of personalized VRML content: a general approach and its application to 3D e-commerce. In: Web3D '02: Proceedings of the seventh international conference on 3D Web technology, New York, NY, USA, ACM (2002) 145–154
29. Estalayo, E., Salgado, L., Moran, F., Cabrera, J.: Adapting multimedia information association in VRML scenes for e-learning applications. In: Proceedings of 1st International Workshop LET-Web3D. (2004) 16–22

30. Dachselt, R., Hinz, M., Pietschmann, S.: Using the AMACONT architecture for flexible adaptation of 3D web applications. In: Web3D 06: Proceedings of the eleventh international conference on 3D web technology, New York, NY, USA, ACM (2006) 75–84
31. Hinz, M., Fiala, Z.: Amacont: A system architecture for adaptive multimedia web applications. In Tolksdorf, R., Eckstein, R., eds.: Berliner XML Tage, XML-Clearinghouse (2004) 65–74
32. Dachselt, R., Hinz, M., Meissner, K.: Contigra: an XML-based architecture for component-oriented 3D applications. In: Web3D '02: Proceedings of the seventh international conference on 3D Web technology, New York, NY, USA, ACM (2002) 155–163
33. Chittaro, L., Ranon, R.: Adaptive hypermedia techniques for 3D educational virtual environments. IEEE Intelligent Systems **22**(4) (2007) 31–37
34. Chittaro, L., Ranon, R.: An adaptive 3D virtual environment for learning the X3D language. In Bradshaw, J.M., Lieberman, H., Staab, S., eds.: Intelligent User Interfaces, ACM (2008) 419–420
35. Bra, P.D., Aerts, A., Berden, B., de Lange, B., Rousseau, B., Santic, T., Smits, D., Stash, N.: AHA! The adaptive hypermedia architecture. In: HYPERTEXT '03: Proceedings of the fourteenth ACM conference on Hypertext and hypermedia, New York, NY, USA, ACM (2003) 81–84
36. Métral, C., Falquet, G., Vonlanthen, M.: An ontology-based model for urban planning communication. In Teller, J., Lee, J.R., Roussey, C., eds.: Ontologies for Urban Development. Volume 61 of Studies in Computational Intelligence. Springer (2007) 61–72
37. Bilasco, I., Villanova-Oliver, M., Gensel, J., Martin, H.: Sémantique et modélisation des scènes 3D. RSTI-ISI, Metadonnées et nouveaux SI **12**(2) (2007) 121–135
38. Bilasco, I.M., Villanova-Oliver, M., Gensel, J., Martin, H.: Semantic-based rules for 3D scene adaptation. In Gervasi, O., Brutzman, D.P., eds.: Web3D, ACM (2007) 97–100
39. Bilasco, I.M., Gensel, J., Villanova-Oliver, M., Martin, H.: An MPEG-7 framework enhancing the reuse of 3D models. In Gracanin, D., ed.: Web3D, ACM (2006) 65–74
40. Barroca, B., Amaral, V., Risoldi, M., Caprini, M., Moreira, A., Araújo, J.: Towards the application of model based design methodology for reliable control systems on HEP experiments. In: 11th IEEE Nuclear Science Symposium, Proceedings, IEEE (10 2008) URL `http://www.nss-mic.org/2008/NSSMain.asp`.