A UML PROFILE FOR CODE GENERATION OF COMPONENT BASED DISTRIBUTED SYSTEMS

G. Chiozzi, R. Karban, L. Andolfato, ESO, Garching Germany
A. Tejeda, Universidad Católica del Norte, Antofagasta, Chile

Abstract

A consistent and unambiguous implementation of code generation (model to text transformation) from UML must rely on a well defined UML profile, customizing UML for a particular application domain. Such a profile must have a solid foundation in a formally correct ontology, formalizing the concepts and their relations in the specific domain, in order to avoid a maze or set of wildly created stereotypes. The paper describes a generic profile for the code generation of component based distributed systems for control applications, the process to distil the ontology and define the profile, and the strategy followed to implement the code generator. The main steps that take place iteratively include: defining the terms and relations with an ontology, mapping the ontology to the appropriate UML meta-classes, testing the profile by creating modelling examples, and generating the code.

INTRODUCTION

In every phase of a new project, one recurring activity is the assessment of artefacts of previous projects for opportunities of re-use. This might include requirements, architectural principles, design and, eventually, code.

Unfortunately, it becomes often apparent that the documentation is poorly articulated or outdated. The knowledge acquired during implementation and commissioning is just reflected in the code and can be "rediscovered" only by reverse engineering.

The developed code is often tightly linked to a specific infrastructure, which might have become obsolete or cannot be adopted due to project constraints.

To overcome some of those deficiencies, platform dependent information needs to be segregated from the domain specific application, and yet it must be possible to keep the two parts aligned automatically.

This principle can be put into practice, using technology which has become recently mature and, robust, such as Model Driven Development [1].

We want to use this type of development at ESO for new projects and the upgrade of existing systems.

MODEL DRIVEN DEVELOPMENT

The availability of open standards like UML, MOF, CWM (www.omg.org/mda) and open tools, frameworks and transformation languages (www.eclipse.org/emf) is a sign that Model Driven Development technologies are becoming mature and widely adopted [2].

In this approach, the system to be developed is described using a precisely defined domain specific modelling language. The artefacts of the development activity (like documentation and code) are derived from the original model using model to model or model to text transformations. The process can consist of multiple layers, each of them moving from an abstract to a more concrete representation of the system or from domain specific to general purpose languages. At the end of the process, the actual application source code is generated.

Up to a certain level of detail, graphical notations are best suited for system modelling [3]. We have therefore adopted the Unified Modelling Language (UML - www.omg.org/uml) as the general purpose modelling language. By now, UML is accepted by engineers in several fields, also outside the software domain.

Moreover, there are tools available, supporting Model Driven Development based on UML, providing means to define modelling languages and model transformations starting from UML.

It is important to notice that this is a step forward with respect to *Model Based Development*, where models are used just in the early phases of a project and afterwards the code is written by hand. *Model Based Development* suffers from the non formal relation between model and code, which causes obsolescence of the models and misalignment with the code.

THE PROCESS

In order to be able to formally define and implement the Model Transformations that allow going from the model to the final code, it is necessary to have a non-ambiguous description of the specific domain concepts.

UML provides a powerful mechanism to express domain specific concepts: the *Profile*.

A *UML Profile* is a collection of definitions for *stereotypes*, *tags* and *constraints* that customize UML for a domain, redefining the semantics of the modelling language in an additive way. This means that, inside a model, different but compatible UML profiles can be used at the same time to specify different aspects of the system.

We decided to develop a UML Profile to capture the meta-model necessary to describe our typical architecture of a telescope and of its instrumentation.

We started with a comparative analysis of the requirements and architecture of projects developed in the last years. From this analysis a first version of the profile was produced.

Unfortunately, while building the profile we realized that there were very limited possibilities to verify automatically the correctness and un-ambiguity of our models with respect to the meta-model. As a matter of fact the only type of validation allowed is based on the definition and verification of UML constraints, which are still not properly supported by most UML tools.

A proper level of formality can be obtained by defining an *ontology* [4] for the domain, i.e. a formal description of the concepts and of their properties, features, attributes, restrictions and relations.

We created an ontology using UML class diagrams, starting from the ones we used to identify the profile elements (Figure 1). However, those concepts and their relations should be expressed in an *ontology description language* like OWL [4].

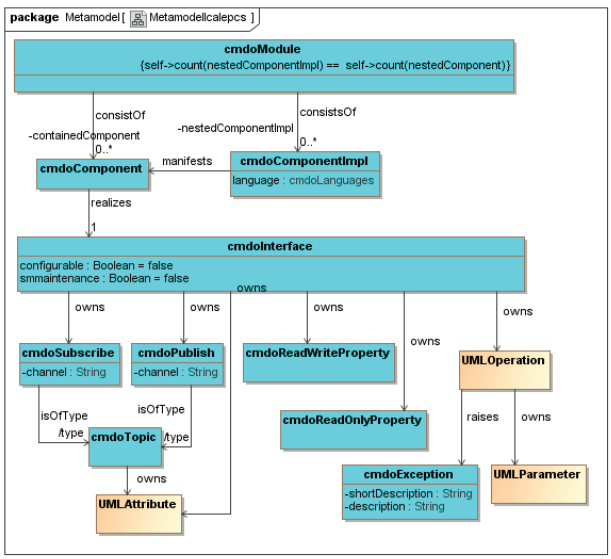

Figure 1: Snippet of the ontology as UML metamodel.

The resulting ontology is heavily affected by our experience and our specific focus, therefore groups with different background might create a different set of concepts to model the same domain.

Depending on the sophistication of the model transformations and of the tools used, we can get different approximations to the real final application.

We have adopted an iterative process consisting of the following steps:

- Analyze and compare parts of real systems we have implemented
- Describe in the *ontology* the common concepts
- Map the ontology to a profile
- Model the system with the new elements
- Implement the transformations
- Compare the generated code with the original code: it must be readable and reliable and it must be possible to add easily hand-crafted code
- Analyze what needs to be modelled and/or refactored at the next iteration

In this work we apply a few basic practical rules:

- An element of the ontology should be mapped in the profile only if it is worthwhile to model, i.e. if it takes less effort to transform the model representation with respect to an handcrafted implementation. The gain should consider parameters like one-to-many transformation scenario, copy&paste errors, or the number of times the element appears in our models.
- If a concept of the ontology applies to all model elements, it should be implemented via an external parameter passed to the transformation engine, while if it applies only to some model elements, it should be a stereotype or a tag.
- If a concept of the ontology applies only to some of the supported platforms, the transformation is ignored for the other platforms.

In the beginning we must expect to add code by hand for substantial parts of the application on top of a simple auto-generated skeleton, but an iterative process of extensions will add more and more automation.

The selection of the transformation language plays an important role in the definition of the profile, because some languages may provide better support for some profile element than others (for example, trading off between using stereotypes or tags favours stereotypes when using the Xpand language). We selected EMF and the Xpand tool set [5] because already used at ESO for some other projects. More details on the code generation process and its implementation are provided in [6].

THE COMODO UML PROFILE

In the overall system model, which includes the representation of hardware components and is typically defined using the System Modelling Language (SysML) profile, our new COMODO (COmponent MODelling for Observatory) profile allows to and model the parts used to generate the code of the real control system.

By keeping the abstract and Platform Independent information separated from the concrete Platform Specific information, we can generate application code for the same system on different platforms by simply replacing the model-to-model transformation for this last layer. A significant effort was spent in separating the Platform Independent Model (PIM) from the Platform Specific Model (PSM).

We will describe here below and in Table 1 some profile elements.

Distributed Components

Telescope control systems fit very well in an architecture based on Distributed Components, i.e. in an architecture where independent entities (called Components) communicate concurrently and asynchronously with each other.

In order to keep a clear distinction between PIM and PSM, we identified three entities in our meta-model:

- **Interface (cmdoInterface - PIM)**
- **Component (cmdoComponent - PIM)**
  In particular, state charts are used to specify behaviour. As described with more details in [6], state charts have been successfully applied at ESO for modelling reactive systems. An extensive usage as a core element of design, supported by code generation, simplifies the development.
- **Component Implementation (cmdoComponentImpl - PSM)**

Table 1: Basic Elements of the Comodo Profile

Name	Base Classifier	Documentation
cmdo Interface	UML Interface	Part of: PIM Public interface that the Component is exposing. Can include: • Operations provided (synchronous invocation paradigm) • Published and subscribed data structures (asynchronous communication paradigm) • Properties, to represent data attributes that can be read/set/monitored • Error handling and exceptions specification • Signals to be used as triggers for the state machine
cmdo Component	UML Class	Part of: PIM Platform independent representation of a Component as it is going to be implemented, realizing the corresponding Interface and optionally specifying the dynamic behavior with a state chart. A design including algorithms or private attributes/operations used in any implementation can be specified here.
cmdo ComponentImpl	UML Artifact	Part of: PSM The Component Implementation is a manifestation of the corresponding cmdoComponent. It represents the actual implementation for a specific platform. If we have multiple implementations for the Component, for example with different programming languages or for different infrastructures, we can model each of them independently without affecting the platform independent representation.
cmdo Topic	UML Signal	Part of: PIM A Topic is a type for (published or subscribed) attributes or parameters. Used as triggers by State Machine.
cmdo Publish	UML Property	Part of: PIM Used to specify whether an attribute in the interface whose type has stereotype cmdoTopic has to be published.
cmdo Subscribe	UML Property	Part of: PIM Used to specify whether an attribute in the interface whose type has stereotype cmdoTopic has to be subscribed.
cmdo Module	UML Package	Part of: PSM This is an organizational unit for specifications of components, interfaces, etc. It identifies a set of deliverables.
cmdo Machine	UML Node	Part of: PSM Machines represent the physical system were one or more containers can run. The deployment of containers is modeled as attributes of machines.
Cmdo Exception	UML Signal	Part of: PIM Exception are used to model errors returned by the interfaces.
Cmdo Container	UML Exe.Env	Part of: PSM Containers are the environments where one or more component implementations can run. The deployment of component implementations is modeled as attributes of containers.

Deployment

The deployment information for the system is part of the PSM, since the same architectural Components can be deployed in different ways depending on the platform.

Moreover, we can have different deployments for the same applications in different phases of the project or for different purposes (testing, commissioning or operation). Deployment is typically a separate responsibility from development and is done by different actors (*system configurators* with respect to *developers*). Our meta-model assumes that Components are deployed in Containers, according to the Component/Container paradigm used by many distributed infrastructures.

A Container provides to Components a set of services (like logging or Component location) that helps shielding the implementation of Components from the specific software infrastructure adopted. In the simplest case of the VLTSW infrastructure the Container is just implemented with a CCS Environment [7], while in the case of ACS a Container is a process that can dynamically load and host multiple Components [8].

E-ELT PROTOTYPE INSTRUMENT

As an example, we show here the model corresponding to a small instrument described in [9], which was developed as a prototype for the evaluation of software and electronics for E-ELT instrumentation (Figure 2).

In this model we define generic interfaces for Subsystems and Devices (Figure 3) which are specialized for a particular application. For example, from the Device we specialize Motors and from that Filter Wheels.

All Devices share a state machine specified in the Device Component platform independent definition.

The model includes the implementation specification for all these components in Java and a test deployment.

With the existing code generator, we can produce from this model a skeleton implementation of this system for the ACS platform, providing sufficient functionality to start-up the system and test interfaces and state machines.

The developer can immediately implement the behaviour code (such as state machine actions and activities) inside the skeletons. With the help of an application framework, a limited knowledge of the

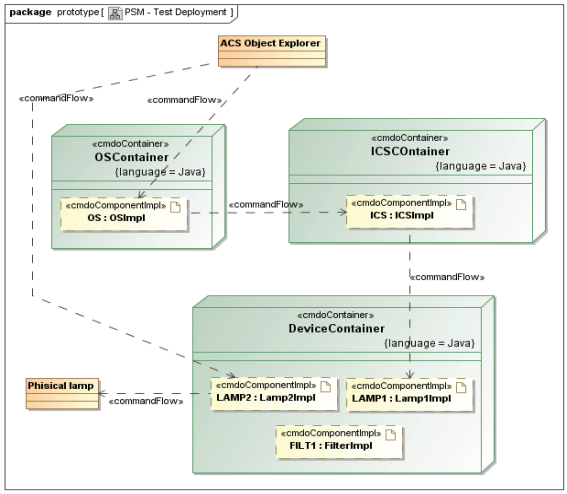

Figure 2: Partial deployment of the instrument.

specific platform is sufficient to complete the development, flattening considerably the learning curve.

In the case of changing platform, all of the PIM of the model would remain the same and probably (depending on the platform) also a big part of the PSM.

Definitions like the ones for Device or Motor are very generic and could therefore become part of the Profile itself or of a domain/project specific extension.

FUTURE DEVELOPMENT

Both, the ontology and the profile need to be extended to cover more domain (telescope and instrumentation) specific aspects of our systems. At the current stage, we have just a generic component/container distributed system model, but we have started to define an architecture framework for telescopes and instruments.

This will be specified through an ontology, with a proper ontology specification language, to allow formal checking.

On the code generation side, we are working on extending the support for ACS to the C++ language. This is also the first step to support the VLT platform, which is essentially based on C++.

CONCLUSIONS

This work is helping us significantly in reaching the objective of modelling our systems in a platform independent way, leaving big parts of the code production to code generators. In this way, the same model can be reused for different infrastructures and, therefore, for different projects and with a longer life span. At the same time, developers are shielded from the details of the specific platform.

This allows us to work now on the modelling of E-ELT TCS and instrumentation without knowing what infrastructure will be finally used, but testing our architecture on the ACS platform (or on the VLT once the transformations will have been implemented).

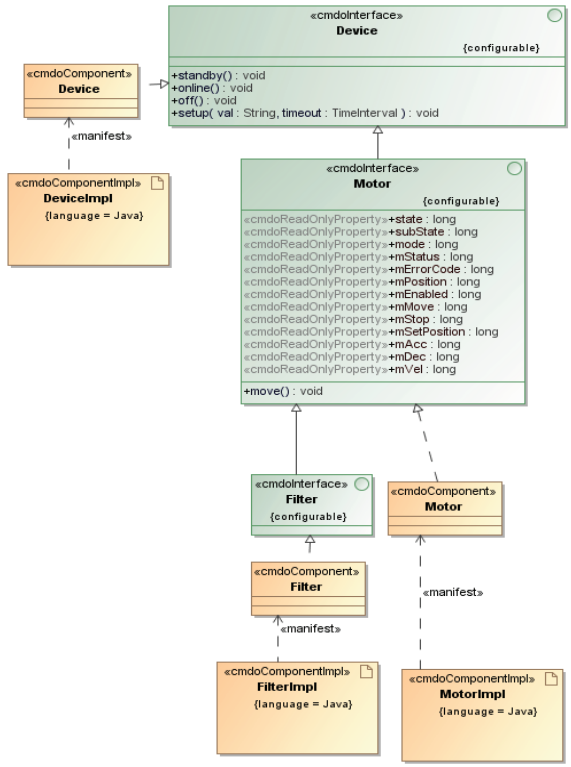

Figure 3: Specifications for motors in the prototype.

REFERENCES

[1] S. Staab et al., "Model Driven Engineering with Ontology Technologies", Reasoning Web: Semantic Technologies for SW Engineering (2010)

[2] G. Booch, et al., "An MDA Manifesto", MDA Journal, (2004)

[3] D. Harel, "Statecharts in the Making: A Personal Account", Communications of the ACM, 03/2009, Vol. 52, No.03

[4] N. F. Noy, D. L. McGuinness, "Ontology development 101: A guide to creating your first ontology", Tech. Rep. SMI-2001-0880, 2001

[5] B. Klatt, "Xpand: A Closer Look at the model2text Transformation Language" 12th European Conference on Software Maintenance and Reengineering, (2008).

[6] L. Andolfato et al., "A Platform Independent Framework for Statecharts Code Generation", this conference, (2011).

[7] M.J. Kiekebusch et al., "Evolution of the VLT instrument control system toward industry standards", Proc. SPIE 7740, 77400T (2010)

[8] G. Chiozzi et al., "ALMA Common Software (ACS), status and development", in [Proceedings of ICALEPCS] (2009)

[9] P. Di Marcantonio et al., "Evaluation of Software and Electronics Technologies for the Control of the E-ELT Instruments: a Case Study", this conference (2011)