Experiences in Applying Model Driven Engineering to the Telescope and Instrument Control System Domain

Luigi Andolfato, Robert Karban, Marcus Schilling, Heiko Sommer, Michele Zamparelli, and Gianluca Chiozzi

European Southern Observatory, Karl-Schwarzschild-Str. 2, Garching bei München, Germany
{landolfa,rkarban,mschilli,hsommer,mzampare,gchiozzi}@eso.org

Abstract. The development of control systems for large telescopes is frequently challenged by the combination of research and industrial development processes, the bridging of astronomical and engineering domains, the long development and maintenance time-line, and the need to support multiple hardware and software platforms. This paper illustrates the application of a model driven engineering approach to mitigate some of these recurring issues. It describes the lessons learned from introducing a modeling language and creating model transformations for analysis, documentation, simulation, validation, and code generation.

Keywords: model driven engineering, telescope control systems, model transformation, model validation, code generation.

1 Introduction

1.1 The European Southern Observatory Programmes

The European Southern Observatory (ESO) is an intergovernmental astronomy organization that carries out ambitious programmes focused on the design, construction and operation of observing facilities. ESO has its headquarters in Garching bei München (Germany) and operates three observing sites in Chile: La Silla, Paranal, and Chajnantor. The two major programmes of ESO during the last 20 years were the Very Large Telescope (VLT) and the Atacama Large Millimeter Array (ALMA).

The VLT [16] is an optical-light astronomical observatory and consists of an array of four telescopes, each with a main mirror of 8.2m diameter, that can observe together or individually and four smaller (1.8m) telescopes dedicated to interferometry, making it the largest facility of its kind. The construction of the VLT started in 1988 and it has been fully operational at the Paranal Observatory since the year 1999.

ALMA [17] is a global partnership between the scientific communities of East Asia, Europe and North America with Chile. It comprises an array of 66 12-metre and 7-metre diameter antennas observing at millimeter and sub-millimeter wavelengths. Its construction started in 1998 and in early 2013 it was handed over to the science operations at the Chajnantor site.

J.Dingel et al. (Eds): Model-Driven Engineering Languages and Systems
Lecture Notes in Computer Science Volume 8767, 2014, pp 403-419
Springer International Publishing Switzerland 2014

1.2 Telescope and Instrument Control Systems

An astronomical observation consists of collecting electromagnetic radiation (such as visible light) emitted or reflected from a distant celestial target. Optical telescopes collect the light. Instruments create images analyzed for intensity, size, morphology, or spectral content. Telescope and instruments form a tightly coupled system [19].

Control systems for astronomical observing facilities execute observing blocks, defining celestial targets, necessary boundary conditions (e.g. required atmospheric conditions), and observing modes (e.g. quality of the wave front) to produce scientifically-relevant data. The Telescope Control System (TCS) main goal is to maintain wave front or radio signal quality throughout the duration of the observation. The Instrument Control System (ICS) is responsible for acquiring the scientific data using the TCS to receive the wave front. The TCS includes all hardware, software, and communication infrastructure required to control the telescope and the dome. It provides access to the opto-mechanical components, manages and coordinates system resources, and performs fault detection and recovery. Large observing facilities involve the control and coordination of distributed actuators and sensors, the real-time compensation of atmospheric turbulences, and the coordination of the safety functions to protect humans and the system itself from hazardous situations.

When building control systems for large science facilities, like telescopes, a number of challenges have to be faced. Telescopes and their instruments are interdisciplinary and software intensive systems with long operational life-times between 10 and 50 years. While two generations of telescopes are typically 15 years apart, introducing major technological changes, new instruments are introduced every year and are bound to the telescope's technology.

Although they are one-of-a-kind experimental machines with many components that had never been built before (e.g. nanometer accuracy position actuators, very low noise CCDs), they have to guarantee high dependability. For example the VLT requires maximum technical downtime of 3% during the observation time.

Most of the time, it is not possible to perform complete system tests before the deployment in the operational environment. This is due to different ambient and observing condition constraints and to the cost of integrating the full system which can be afforded only once. Therefore the architecture needs to build in the capability to cope with last minute changes such as modifications in the control system hierarchy, different combination of actuator and sensors, different interaction of distributed control loops.

Despite the fact that those systems are at some point handed over to science operations, they are never frozen but evolve over their lifetime. New scientific objectives may require additional functionalities, or hardware and software can become obsolete. The result is a telescope with subsystems and instruments running on different control SW releases or even different versions of hardware and software infrastructure. A key element of the software infrastructure is the software platform which is used to develop the control applications and includes operating systems, programming languages, communication middleware, IDEs, application frameworks, real-time database, logging, alarms, configuration and error handling services. The overview of the software platforms used for various ESO programmes is given in Table 1.

Table 1. Software platforms at ESO.

Software Platform	*Programme*	*OS*	*RTOS*	*Languages*	*Middle-ware*
VLTSW	Very Large Telescope	Linux	VxWorks	C, C++, TCL/TK	Proprietary messaging system
ACS	Atacama Large Millimeter Array	Linux	Linux RT	C++, Java, Python	CORBA, DDS
SPARTA [34]	Very Large Telescope	Linux	VxWorks	C, C++, Java, TCL/TK	CORBA, DDS
Rapid Prototype	Any	JVM	N/A	Java	RabbitMQ [31]

2 Modeling Environment

2.1 Evolution of Modeling Environment

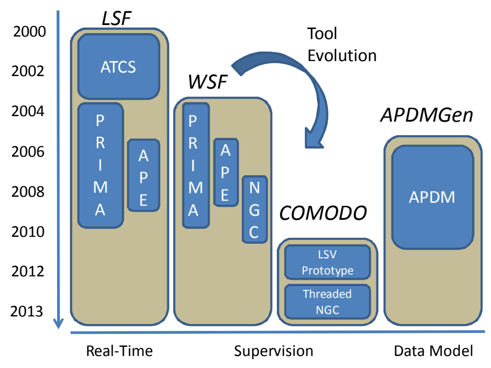
The first successful attempt[1] to apply model transformation to the development of telescope control software was the Local control unit Server Framework (LSF) tool. LSF was created in 2000 to help building the applications running on the real-time local control units providing access to HW. In order to build an LSF application, a configuration file containing information on the number and type of devices to control is processed by a Tcl script which produces the skeleton code of an application with call-backs for custom code to be completed by the developer. In addition, LSF provides a predefined state machine implementation where the developer can hook in code for predefined actions. An LSF application can be extended by adding more device definitions in the configuration files and reapplying the transformation. LSF has been extensively used for the development of the Auxiliary Telescopes Control Software (ATCS), Phase Reference Image and Micro-arcsecond Astrometry (PRIMA) control software, and the Active Phase Experiment (APE).

In 2004, inspired by LSF, a tool suite called Workstation Software Framework (WSF) was developed to generate soft real time supervisory applications [1]. WSF was initially created to build the supervisory applications of the PRIMA control software and later successfully adopted for the development of applications for many other projects of the Very Large Telescope program such as the Interferometric Supervisor Software configuration process, the Delay Lines rail-alignment tool, the APE project, and the New Generation CCD (NGC). In the beginning WSF applications

[1] Earlier, the use of Rhapsody code generation capabilities was investigated and considered too constraining because of the dependency on proprietary run-time libraries.

were generated from a configuration file containing the textual description of their behavior in the form of a state machine. Later on, tools were developed to transform Rational ROSE and MagicDraw UML State Machine models into the text configuration file. The modeling tools acted as a front end to facilitate the creation of Statecharts [10] since, with increasing complexity the maintenance of the text description became significantly more time-consuming. The applications generated by WSF were based on the State design pattern [11].

A development parallel to WSF was started in 2004 for the ALMA programme. The ALMA Project Data Model generator (APDMGen) generates, at the beginning only from XML schema and later also from UML class diagrams, the data classes representing the data model: complex data structures to describe science targets, calibrations, data quality requirements, or hardware configurations.



Project	Applications Size in kSLOC	UML Elements	FTE
ATCS	104	0	14
PRIMA	204	1416	14.4
APDM	113	1680	6
APE	215	8193	17.35
NGC	51	560	3
LSV Prototype	20	1288	0.4
Event Subscriber	1	100	0.1
Threaded NGC	30	1426	0.2

Fig. 1. The left panel shows the projects (blue boxes) developed using code generation tools (brown boxes) in the last 13 years (Y axis). The type of applications built is also reported (X axis). The right panel lists the projects and some information on size and cost: the second column provides the application size in kSLOC; the third column is the size of the input model in terms of UML Elements and their subclasses with the exception of profiles and libraries; the last column is the effort expressed as Full Time Equivalents (FTE), including application specific (meta-) modeling.

In 2009, based on the experience gained with WSF, a new project was started to create a platform independent transformation tool to develop state machine driven applications. Two new main requirements were introduced: the possibility to support multiple software platforms[2] like the control software for the Very Large Telescope (VLTSW) and the Alma Common Software (ACS), and the ability to interpret state machines. The first requirement focused on enabling model reusability across different platforms allowing the developers to create applications regardless of the target development and execution environment. The second requirement aimed at reducing the size of the generated applications by decoupling the application from the state

[2] Note that LSF and WSF tools are specific to the VLT platform and APDMGen works only for the ACS platform.

machine execution engine and to provide the capability of changing the state machine logic at runtime allowing for fast last minute changes. The project delivered a toolkit, called COMODO [2], which has been used to develop the Telescope Control Local Supervisor (LSV) prototype running on a rapid prototyping software platform based on Java and RabbitMQ [31], to redevelop a new multi-threaded version of the NGC for the VLTSW platform, and to create the Event Subscriber application for the ACS platform. **Fig. 1** summarizes the evolution of the code generation tools at ESO and provides an idea on the size and cost of the projects.

In order to maximize the return on investment of modeling, more applications of model transformations were explored in addition to the ones targeted on the final production code. For example model simulation was used to get an early feedback on the logical correctness of the model especially in the context of collaborating state machines. Initially simulation was applied in order to understand some principles of State Analysis methodology [15] and later on to verify the behavior of telescope control architecture. However, it became quickly clear that proper model validation could be better achieved using a model checking approach. Therefore COMODO was extended to support a transformation to the Java Pathfinder model checker to be able to formally validate state machine models [6]. This transformation was applied to validate the control software design of the PRIMA Variable Curvature Mirror and, in collaboration with NASA/JPL, to verify part of the Soil Moisture Active Passive fault protections system [21].

To guarantee consistency between models and documentation some effort was spent in 1999 to implement a "one document" approach [36] where HTML and Word documents were produced using Telelogic DocExpress from Rational ROSE models. Unfortunately the transformation framework offered insufficient control over the generated artifacts and therefore this approach was used only in the ATCS project. Ten years later a plug-in for MagicDraw, the Model Based Document Generator [18][24], was developed in-house with ownership over the transformation allowing full compliance with ESO documentation templates.

Finally, the recent Conceptual Modeling Framework (CMF) initiative aims at enforcing model correctness using ontologies to capture more formally business rules.

2.2 Current Status

The modeling environment currently in use is based on the following elements.

UML™ / SysML™ modeling languages and MagicDraw®. MagicDraw [29] is a commercially available software and system modeling tool with teamwork support. It supports UML 2 [27] and, via plug-in mechanism, SysML [28]. The Cameo Simulation Toolkit® [30] is a plug-in for MagicDraw which provides an extendable model execution framework based on OMG fUML [35] and W3C SCXML [4] standards.

Conceptual Modeling Ontology. The Conceptual Modeling Ontology (CMO) is an ontology language similar to OWL2 [33] introduced, in form of UML Profile, to permit the expression of business specific concepts and relationships recurring across all our models. It has been developed by ESO based on work done by NASA/JPL [23] and some experiences in defining DSLs using SysML [24]. CMO is also used to express the mapping between the ontology and the UML meta-model elements. Various layers of interdependent ontologies are supported.

Conceptual Modeling Framework. The Conceptual Modeling Framework (CMF) is an approach, under development at ESO, for turning UML into a domain specific modeling language. It transforms ontologies written in CMO into UML profiles, the associated validation rules and custom diagram editors. The generated validation rules are used by MagicDraw's validation engine which can run on-demand or can constantly check the model in the background while it is being edited. MagicDraw customization features are used to adjust the diagram editor to only offer certain element types to the modelers according to the specified ontology.

Model Based Document Generator. The Model Based Document Generator (MBDG [18][24]) is a profile and a plug-in for MagicDraw developed by ESO to be able to write documents as SysML models and to transform them into DocBook [25] XML files. Since documents and system models coexist within the same modeling environment, duplication of information is avoided and consistency is automatically maintained. The generated DocBook files can be converted into different document formats such as PDF.

APDMGen. The ALMA Project Data Model generator is a toolkit, developed by ALMA and based on openArchitectureWare [22], to transform UML class diagrams into XML schemas and Java data classes.

Java Pathfinder model checker. Java Pathfinder (JPF) [32] is a system to verify executable Java byte code programs. JPF was developed at the NASA Ames Research Center and open sourced in 2005. It provides an extension, called jpf-statechart [6], used to execute and systematically verify Statecharts models.

SCXML Engine. The SCXML engine is required to interpret the SCXML documents that describe applications behavior. For Java applications, the Apache Commons SCXML [5] is used, while for C++ the scxml4cpp library has been developed by ESO. The Apache Commons SCXML is also used by Cameo Simulation Toolkit.

COMODO Ontology and Profile. The COMODO ontology, based on CMO, captures the concepts and relations required to describe the structure and behavior of component based distributed systems. The COMODO profile is the UML representation of the COMODO ontology [3]. COMODO ontology and profile have been developed by ESO to be used by the COMODO Toolkit.

COMODO Toolkit. COMODO Toolkit transforms UML models, based on the COMODO profile, into different artifacts depending on the target platform. In addition to the VLTSW, ACS, and Rapid Prototype software platforms, it supports plain Java, and Java Pathfinder model checker by generating Java code compliant with jpf-statechart[3]. A summary of the artifacts and activities involved in a COMODO transformation is given in **Fig. 2**. For all target platforms, the input model, together with some configuration information such as the part of the model to transform and the target platform itself, is transformed by COMODO into:

- One or more application skeletons.
- One SCXML document compliant with the StateChartsXML notation defined by the W3C [4] for each UML State Machine[4]. The mapping between UML and SCXML has been defined in [2].
- Test code.
- Build files (ant or makefile).

The generated artifacts together with the developer's implementation of the actions and do-activities are compiled and linked with platform specific libraries such as the SCXML engine (Apache Commons SCXML library [5] or scxml4cpp library).

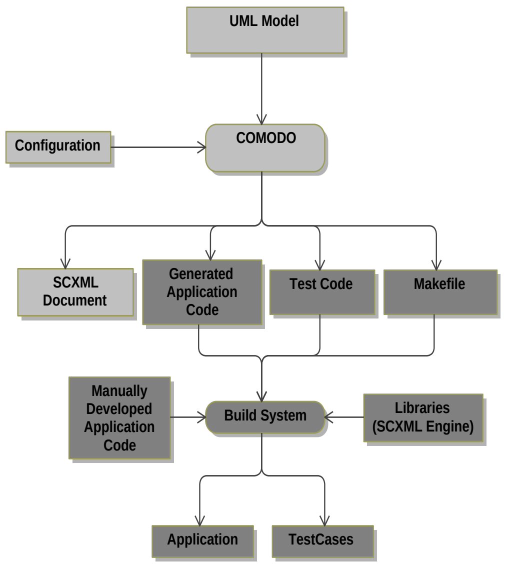


Fig. 2. COMODO data flow: in dark gray the platform dependent artifacts and activities; in light gray the platform independent ones.

[3] By inserting manually assertions in entry/exit/transition actions it is possible to verify properties of the system.

[4] For the Java Pathfinder Statecharts platform the SCXML document is not used.

COMODO is composed of a java front-end processing the input parameters and triggering the execution of the modeling workflow (EMF MWE [12]) specific to the target platform, a set of Check model validation rules applied to the UML model, a set of Xpand [13] templates organized by target platform (VLTSW, ACS, etc.) and target language (C++, Java, XML, text, etc.), and a library of Xtend functions to navigate the model (**Fig. 3**)[5].

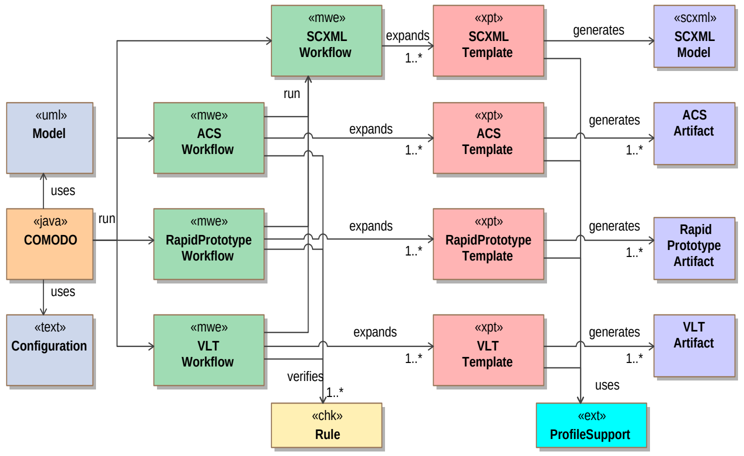


Fig. 3. COMODO's structure. Only four target platforms are shown here: SCXML, VLTSW, Rapid-Prototype, and ACS. SCXML workflow is reused by the other three platforms. Stereotypes indicate the language: mwe = modeling workflow engine, xpt = Xpand, ext = Xtend.

3 Lessons Learned

3.1 Modeling Language

The process to move from traditional programming languages to a more abstract language has been gradual and natural. For example, a developer in charge of building ten or more applications tries to minimize the repetitive work by copying the first application nine times and replacing the application specific parts. The application specific parts are usually composed of concepts that can be abstracted using a modeling language (like states, events, and state transitions) and concepts which are hard to abstract (like the implementation of actions and do-activities). Initially the abstract information was stored in text files using simple property-value syntax or using XML schemas. However for large models we quickly felt the need to use a graphical notation to group parts of the model and emphasize certain view points. In addition, models based on topological concepts (e.g. Statecharts) are easier to appreciate using visual formalism than text [20][7]. Therefore WSF, APDMGen, and COMODO tools

[5] MWE, Check, Xpand, and Xtend were part openArchitectureWare toolkit [22] and are now included in the Eclipse Modeling project [14].

provide the ability to process models created with graphical UML modeling tools. Unfortunately UML and related tools are not as simple and fast to make small changes as a text language and a text editor. They require some skills which can be easily forgotten if the tools are used only once or twice per year (as it may be the case during the software maintenance phase). Two opposite needs have been observed:

- During development, when models have to be frequently changed, reviewed and discussed, the graphical representation of models containing topological information is very important since it is easier to understand and more compact.
- During maintenance, the software has to be modified few times per year. The maintenance engineers are infrequent users of our graphical tooling. They feel they can apply small changes to the model much more quickly in a text editor.

At the moment, we support both textual (SCXML, XML schemas) and graphical representation (UML Class diagrams and State Machine models) as successfully adopted in WSF in the past. In order to avoid diverging of the two representations, changes to the textual one must be recorded and ported back to the graphical one. The lesson learned is that we need to restrict and customize the user interface of the modelling tool to provide only a subset of UML specialized on Statecharts and composite structures.

3.2 Obsolescence Management of Tool Chain

Due to the long development and operational life time, the obsolescence of a third-party tool chain and the associated competence is a major concern. A number of risks have been identified, in particular for modeling and model transformations activities. They are related to the unavailability and/or change of:

- UML/SysML modeling tool (MagicDraw and plug-ins)
- UML profile and meta-model (EMF UML2)
- transformation languages (Xpand, Xtend) and modeling frameworks (EMF)
- competent people

The first UML tool integrated with WSF has been RationalROSE by Rational (now IBM). Later on MagicDraw became our standard UML tool. The porting of the models from RationalROSE to MagicDraw was done manually since the automatic export/import procedures to XMI did not work, since UML meta-models were different. Despite the Model Interchange Working Group effort [37][6], still today it is a challenge to port models between different UML commercial tools. This is definitely a problem since large telescopes have more than 20 years life time and we cannot afford to rely only on a single tool vendor or to manually port large models. At the mo-

[6] The test case examples proposed are trivial and cover only a subset of UML. For example, in the State Machine examples history states, internal transitions, nested orthogonal regions, and different types of triggers and behaviors are not covered.

ment, in order to avoid vendor lock in, COMODO supports UML models in the EMF UML2 XMI format.

Changes in the meta-model bear the risk of corrupting existing models because sometimes the migration path from one version to another is not well defined. In particular when meta-model elements disappear, as it happened from UML 2.3 to 2.4 where the ExecutionEvent event type became obsolete causing potential data loss (avoided by developing an ad-hoc M2M transformation).

An intermediate vendor independent representation of the model (e.g. EMF XMI and SCXML) is used to mitigate the risks associated to the modeling tools and UML. Transformation languages and modeling frameworks were selected among the ones with larger user base, open source, and most compliant with standards. Concerning the competences, a small team with modeling and model transformation know how was established. The team is in charge of providing modeling support to the projects and customizes the transformations.

3.3 Transformation Ownership

A key point in the successful adoption of MDE is the ability to customize the transformations to have full control over the generated artifacts [8]. This allows:

- generating code conforming to project standards, guidelines and platforms
- producing documentation using the organization's templates
- supporting changes to the meta-model
- managing problems downstream the tool chain such as new versions (or deficiencies) of libraries and compilers

3.4 Platform Independent Modeling

During the creation of COMODO, the definition of the ontology has been the most time-consuming activity. We believe that this is a general issue since the ontology definition is an iterative process involving domain specialists capturing the necessary semantics to enable an efficient and correct transformation. The ontology had to be adapted many times before a stable compromise between formality and practicability could be found. The UML profile resulting from the ontology is platform independent and is designed to be used for all ESO target platforms[7]. Platform specific information, when needed, is provided directly to the model-to-text transformation tool via command line arguments or a configuration file. This approach intentionally avoids the Platform Independent Model (PIM) to Platform Specific Model (PSM) model-to-model transformation, suggested in [26], since this introduces not negligible development and maintenance costs, especially when dealing with UML as target meta-model. For example, the type of target platform is given as a configuration parameter to the tool and does not appear in the model.

[7] Features appearing in the meta-model and semantically irrelevant for a specific target platform are ignored by the transformation tool.

3.5 Modeling vs. Coding

An important lesson learned from WSF development relates to the amount of generated code. Even though model-to-text transformations take usually an insignificant amount of time, the compilation of the generated code can be time-consuming. Therefore it is important to be able to transform only part of the model: in this way we avoid rebuilding the whole system at each modification. Moreover, preference should be given to the usage of configurable libraries instead of code generation. For example, control applications created with WSF are based on the State Design pattern which requires the generation of one C++ class per state while applications created by COMODO use a state machine engine library able to execute SCXML documents. In the latter case only the SCXML description of the state machine has to be generated. In general our transformations are targeted for "rich" software platforms: platforms which include all common services required by the applications (such as logging, messaging, error and alarm handling, configuration management, etc) and do not need to be generated.

3.6 Semantic Consistency

There are different flavors of Statecharts semantic [9] and to avoid inconsistency it is important to stick to one across the tool chain. For example, UML does not specify any language construct to query at run-time the current Statechart configuration ("inState()" or "In()" as defined in [10]). In SCXML the active Statechart configuration is updated after invoking the exit actions and before invoking the entry actions. We chose SCXML's over alternative implementations for the following reasons:

- It provides well defined syntax and operational semantic as pseudo-code
- The relevant features of UML State Machines can be easily mapped to SCXML
- The same engine is used for model simulation, production code, and prototyping

Unfortunately the validation step, currently based on jpf-statecharts, is not following the SCXML semantics.

3.7 Archive Generated Artifacts

Despite the risk of using outdated artifacts, we keep under version control generated code in addition to the models for the following reasons:

- To have quick access to the generated artifacts (e.g. for urgent modifications in an operational environment) and speed-up the build process.
- To verify that models are equivalent by comparing the generated artifacts.

The second point is very important since it avoids having to repeat system tests when models have to be ported to new tools or to evaluate the impact of changes in the meta-model.

We also learned that, when using commercial tools, any floating license server application should be subject to the same version control procedures as the rest of the tool chain. Failing to do so prevented us from running legacy versions of the tool.

3.8 Model Correctness

Due to complexity, sometimes weak semantics and general purpose of UML/SysML it is necessary to customize it and guide the modeler with standardized patterns and conventions. The compliance of the user model with the defined rules can be verified in various ways (e.g. offline analysis). However, we have observed that one effective way is giving the modeler immediate feedback during the modeling activity to create upfront a model which is correct by construction. This can be achieved by reducing the number of choices that modelers can make, prescribe certain modeling patterns, and come up with concise semantics. CMO and CMF are conceived for this purpose. CMO, following the recommendations given in [23], focuses on conveying in UML syntax the logical organization of a conceptual ontology whose essential constituents are unary concepts and reified binary relationships. This approach has the advantage that the ontology and the user model can be modeled with a single tool and the same language (UML/SysML). It has been used to define a number of reusable ontologies: foundational ontologies (Interface Ontology, Structural Ontology), engineering oriented ontologies (Protocol Ontology, Connector Ontology), telescope oriented ontologies (Telescope Instrument Ontology).

3.9 Roundtrip and Annotated Code

From the beginning we avoided round-trip transformations since transforming back the code and merging it into the model is considered too expensive to implement and maintain. Instead, a clear separation of generated code from manually crafted code is preferred. Generated code is stored in dedicated files which can be referenced using delegation or inheritance mechanisms.

Moreover, we observed that is not efficient to model the behavior of actions and activities because it requires the same time (in the best case, since code editing capabilities within the modeling tool cannot compete with a conventional IDE) as writing the target code and introduces additional transformation from UML or platform independent action languages (e.g. ALF) to the target code. If the model is annotated with target code then the model-to-text transformation has to be executed every time the model or the annotated code is modified. In addition the model is not platform independent anymore[8].

[8] In the executable models or model simulation scenario the annotated code is usually a simplification of the final production code. A mapping of simulation code to final production code can be quite challenging.

3.10 Reusable Modeling

Solutions to recurring problems of control application can be extracted in the form of a set of modeling patterns documented and collected in a catalog similarly to Design Patterns. Some examples of state machine modeling patterns are described in [1].

For particular domain specific classes of applications, the whole model is used as a template to be copied and pasted. Certain elements of the model are parameters (e.g. events or actions in state machines) that can be replaced with concrete arguments.

3.11 Cost / Benefit Analysis of Model Transformations

ESO's primary goal is the delivery of telescopes and instruments and not the development of modeling tools. It is therefore important to constantly compare the effort of abstracting information and transforming it into specialized artifacts with the cost of creating the specialized artifacts manually.

Given a generic SW application, it is always possible to find an abstraction of the application, called model, and define its source code as composed of two parts: one that is model dependent (MD) and one that is model independent (MI). A very simple abstraction is, for example, a function name: the model is simply the name of the function. Using this abstraction the function's source code can be separated into two parts: the name of the function (model dependent because it is generated) and the body of the function (model independent because it is hand crafted) without the name of the function.

If TAPPL is the total effort, measured for example in Full Time Equivalent (FTE), spent to develop an application, then:

$$\text{TAPPL} = \text{TMI} + \text{TMD} \tag{1}$$

where:

- TMI = is the average effort spent to develop by hand the model independent part of an application
- TMD = is the average effort spent to develop by hand the model dependent part of an application

A model-to-text transformation requires:

- the definition of a source meta-model (TMMDEF)
- the ability to navigate models based on the source meta-model (TMMNAV)
- the creation of the templates required to generate the target artifacts (TTPL)[9]
- the creation of the model to transform (TM)

Therefore the effort[10] to build N applications using model to text transformation is:

$$\text{TAPPL} = \text{TMMDEF} + \text{TMMNAV} + \text{TTPL} + N * (\text{TMI} + \text{TM}) \tag{2}$$

[9] TTPL includes also the development of libraries used by the templates.

[10] The effort to apply the transformation is considered to be negligible.

The efficiency of developing an application using model to text transformation with respect to developing the application by hand requires the comparison the cost of the two approaches:

$$TMMDEF + TMMNAV + TTPL + N * (TMI + TM) \leq N * (TMI + TMD)$$

$$(TMMDEF + TMMNAV + TTPL) + N * TM \leq N * TMD \qquad (3)$$

The model transformation approach is more efficient if:

$$(TM \leq TMD) \text{ and } (N \text{ is big enough})$$

N has to be big enough so that the fixed cost for the creation of the meta-model (TMMDEF), the development of the tool to navigate the meta-model (TMMNAV) and the templates (TTPL) is absorbed by the difference between writing by hand the model dependent code and creating the model. The model transformation approach tends to be more efficient with simple meta-models easy to navigate and that allow the creation of compact models. Note that, for projects within the same organization, ambiguities in the effort measurement can affect in the same way both terms of eq. 3.

For example, the NGC project, composed of five applications based on WSF, required about 3 FTEs to implement the same functionalities of a similar project (FIERA) which took about 6.9 FTEs. Both projects were done by roughly the same team. The average effort for the NGC model independent part of an application (TMI) equals the total effort minus the effort to build its model; i.e. $(3 – 0.1*5) /5 = 0.5$. The average effort for the NGC model dependent part of an application (TMD) equals the effort to build FIERA minus the model dependent part; i.e. $(6.9/5 – 0.5) = 0.88$. WSF development required for the definition of the meta-model (TMMDEF) about 0.02 FTE and for the development of the parser (TMMNAV) 0.76 FTE. The definition of the templates (TTPL) took 1.76 FTE. Using the simple linear model the breakeven point is reached at N=3.3 so that for every further application we save 0.78 FTEs.

In case of K transformations (3) becomes:

$$(TMMDEF + TMMNAV + \sum TTPL_i) + N * TM \leq N * \sum TMD_{i,} \; i = [1 .. K]$$

And therefore:

$$(TM \leq \sum TMD_{i,}) \text{ and } (N \text{ is big enough})$$

This is similar to (3) except that the sum of the effort to develop the templates for various transformations has to be taken into account.

Note that with modern transformation languages like Xpand, writing templates is, in our opinion, very similar to writing normal code. However the assumption TTPL = TMD cannot be made since TTPL includes some of the effort of generalizing MD.

The maintenance activities like adding new features, fixing bugs, or porting to a newer (version of the) SW platform, can affect the model independent part of the application or the model dependent part. In the former case the cost is the same for both approaches. In the latter case the modification may have to be applied to the meta-model, the templates, or the models. Changes to the meta-model are the most expensive since they can imply modifications of the models, templates and/or the tool to navigate the model. Changes to the templates are more efficient by a factor N-1 (where N is number of applications) with respect to the traditional approach. Changes to the models are in general more efficient since the level of abstraction is higher and dependencies (i.e. side-effects introduced by the change) are more evident.

4 Conclusions

In this paper we have presented our experiences in moving from document and code centric development to a process driven by models. The main focus is on behavioral models because they have turned out to be most beneficial for the telescope and instrument software. The model as a single source of information allows having consistency across different transformed artifacts such as code, documentation, simulation, and analysis. Automatic transformations simplify for a wider audience of engineers the usage of specialized tools without requiring expert skills. In addition models are easier to analyze by model checkers than the final target code, thanks to the higher level of abstraction and reduced computational complexity. However we observed that not everything is worth modeling. Therefore we defined a key performance indicator (as a function of the model dependent and the model independent code) to constantly measure the effort introduced by abstracting information and compare it with the effort required by the traditional development practices.

Large Telescope Control Systems have long operational life-time and are evolving continuously. New scientific instruments are constantly introduced and the obsolete components of HW and SW platforms have to be replaced. The ability of transforming domain specific models into new or upgraded target SW platforms by simply updating templates introduces significant advantages. In contrast to the traditional SW development approach, changes can be propagated across a number of existing applications in a systematic and well defined way. The same type of flexibility is also beneficial when dealing with the last minute changes required during the on-site integration and deployment.

Two major problems have been encountered when applying a model driven development process: the possible lack of semantic integrity and consistency among the produced artifacts, and the shortage of modeling competences during the maintenance activities. The former applies to domain specific ontologies that are mapped to standard modeling languages, and to the structural and behavioral models that are used as a source for simulation, validation and code generation. The latter concerns the ability to maintain generated code in a highly dependable system like a telescope without modeling skills.

Acknowledgments. The authors would like to thank N. Rouquette, S. Jenkins, A. Kerzhner from NASA/JPL for the discussions on the ontological modeling, N. Jankevicius from NoMagic for the collaboration on the Simulation Toolkit, and C. Cumani and A. Balestra for providing the effort measurements on NGC and FIERA projects.

References

1. Andolfato, L., Karban, R.: Workstation Software Framework. In: Proceedings of the Society of Photo-Optical Instrumentation Engineers, Vol. 7019, 70191X-1, (2008)
2. Andolfato, L., Chiozzi, G., Migliorini, N., Morales, C.: A platform independent framework for statecharts code generation. In: Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems (2011)
3. Chiozzi, G., Andolfato, L., Karban, R., Tejeda, A.: A UML profile for code generation of component based distributed systems. In: Proceedings of the 13th International Conference on Accelerator and Large Experimental Physics Control Systems, (2011)
4. World Wide Web Consortium: State Chart XML (SCXML) Working Draft Published. December 6, 2012, (2012)
5. Apache Commons SCXML, `http://commons.apache.org/proper/commons-scxml`
6. Mehlitz, P.: Trust Your Model - Verifying Aerospace System Models with Java Pathfinder. In: Proc. IEEE Aerospace Conf. '08, Big Sky, MT, Mar. 1-8, (2008)
7. Harel, D.: Statecharts in the Making: A Personal Account. In: Communications of the ACM, 03/2009, Vol.52, No.03, p.6, (2009)
8. Wagstaff, K.L., Benowitz, E., Byrne, D. J., Peters, K., Watney, G.: Automatic code generation for instrument flight software. In: Proceedings of the 9th International Symposium on Artificial Intelligence, Robotics, and Automation in Space, (2008)
9. Crane, M.L., Dingel, J.: UML vs. Classical vs. Rhapsody statecharts: Not all models are created equal. In: Software and Systems Modelling, Volume 6, Number 4, (2007)
10. Harel, D.: Statecharts: A visual formalism for complex systems. In: Science of Computer Programming, 8(3):231–274, June 1987, (1987)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, pp. 305-313, (2002)
12. Eclipse Modeling Framework: Modeling Workflow Engine, `https://projects.eclipse.org/projects/modeling.emf.mwe`
13. Klatt, B.: Xpand: A Closer Look at the model2text Transformation Language. In: 12th European Conference on Software Maintenance and Reengineering, (2008)
14. Eclipse Modeling Project, `http://www.eclipse.org/modeling`
15. Ingham, M.D., Rasmussen, R.D., Bennett, M.B., Moncada, A.C.: Engineering Complex Embedded Systems with State Analysis and the Mission Data System. In AIAA Journal of Areospace Computing Information and Communication, vol. 2, No. 12, (2005)
16. Wirenstrand, K.: VLT telescope control software: status, development, and lessons learned. In: Proc. SPIE 2003, vol. 4837, p. 965, (2003)
17. Casasola, V., Brand, J.: The exciting future of (sub-)millimeter interferometry: ALMA. In: in Proceedings of the 54th national meeting of the Italian Astronomical Society, (2010)
18. Model Based Document Generator, `http://sourceforge.net/projects/mbse4md/?source=directory`
19. Bely, P. Y.: The design and construction of large optical telescopes. Springer, (2003)
20. Harel, D.: On visual formalism. In: Communications of the ACM, Vol.31, No.5, (1988)
21. Gibson, C., Karban, R., Andolfato, L., Day, J.: Formal Validation of Fault Management Design Solutions. Presented at the Java Pathfinder Workshop 2013, (2013)
22. Haase, A., Voelter, M., Efftinge, S., Kolb, B.: Introduction to openArchitectureWare 4.1.2. In: Model-Driven Development Tool Implementers Forum (MDD-TIF'07) (co-located with TOOLS 2007), (2007)

23. Jenkins, J., Rouquette, N.: Semantically Rigorous Systems Engineering Using SysML and OWL. In: 5th International Workshop on Systems & Concurrent Engineering for Space Applications, (2012)
24. Karban, R., Zamparelli, M., Bauvier, B., Chiozzi, G.: Three years of MBSE for a large scientific programme: Report from the Trenches of Telescope Modelling. In: Proceeding 22nd Annual INCOSE International Symposium, (2012)
25. Walsh, N.: DocBook 5: The Definitive Guide. O'Reilly Media, April 2010, (2010)
26. Frankel, D.: Model Driven Architecture – Applying MDA to Enterprise Computing. OMG Press, p. 191, (2003)
27. Unified Modeling Language (UML), `http://www.omg.org/spec/UML`
28. System Modeling Language (SysML), `http://www.omgsysml.org`
29. MagicDraw, `http://www.nomagic.com/products/magicdraw.html`
30. Cameo Simulation Toolkit, `http://www.nomagic.com/products/magicdraw-addons/cameo-simulation-toolkit.html`
31. RabbitMQ, `http://www.rabbitmq.com`
32. Java Pathfinder, `http://babelfish.arc.nasa.gov/trac/jpf`
33. OWL 2 Web Ontology Language, `http://www.w3.org/TR/owl2-overview`
34. Fedrigo, E., Donaldson, R.: SPARTA: the ESO standard platform for adaptive optics real time applications. In: Proc. SPIE 6272, (2006)
35. Semantics of A Foundational Subset for Executable UML models (FUML), `http://www.omg.org/spec/FUML`
36. Chiozzi, G., Duhoux, P. Karban, R.: VLTI Auxiliary telescopes: a full Object Oriented approach. In: Proc. SPIE 2000, vol. 4009-03, p. 5, (2000)
37. Model Interchange Working Group (MIWG, `http://www.omgwiki.org/model-interchange/doku.php`