CHESS: A Model-Driven Engineering Tool Environment for Aiding the Development of Complex Industrial Systems

Antonio Cicchetti, Federico Ciccozzi
Mälardalen University, MRTC
Västerås, Sweden
{antonio.cicchetti, federico.ciccozzi}@mdh.se

Silvia Mazzini, Stefano Puri
Intecs SpA
Pisa, Italy
{silvia.mazzini, stefano.puri}@intecs.it

Marco Panunzio, Alessandro Zovi, Tullio Vardanega
University of Padova
Padova, Italy
{panunzio, azovi, tullio.vardanega}@math.unipd.it

ABSTRACT
Modern software systems require advanced design support capable of mastering rising complexity, as well as of automating as many development tasks as possible. Model-Driven Engineering (MDE) is earning consideration as a solid response to those challenges on account of its support for abstraction and domain specialisation. However, MDE adoption often shatters industrial practice because its novelty opposes the need to preserve vast legacy and to not disband the skills matured in pre-MDE or alternative development solutions. This work presents the CHESS tool environment, a novel approach for cross-domain modelling of industrial complex systems. It leverages on UML profiling and separation of concerns realised through the specification of well-defined design views, each of which addresses a particular aspect of the problem. In this way, extra-functional, functional, and deployment descriptions of the system can be given in a focused manner, avoiding issues pertaining to distinct concerns to interfere with one another.

Categories and Subject Descriptors
D.2.10 [Design]: Methodologies

General Terms
Theory

Keywords
Separation of concerns, code generation, back propagation

1. INTRODUCTION
Distributed dependable real-time embedded software systems are becoming increasingly complex due to the demand for extended functionalities, reuse of legacy code and components, diverse application fields, and the multi-site nature of their development process [?]. Solutions tackling such a many-fold challenge usually adopt Model-Driven Engineering (MDE) [?] to master the complexity and size of the problem through abstraction; in turn, each viewpoint by which the system can be reasoned about entails a range of specialised domain-specific techniques dealing with the aspects of interest; e.g., requirements specification, extra-functional modelling, deployment.

Combination of domain-specific approaches alleviates one of the major challenges in MDE: ensuring efficiency in the modelling activities by means of modelling artefacts quality, modelling framework usability, unambiguity (just to mention a few), due to the narrow and specific nature of the languages themselves [?]. However, domain-specificity may degrade flexibility: for instance, concepts falling outside the domain require language updates (and tool migrations) to augment and/or refine language expressiveness. Moreover, integration between different languages and tools has to be carefully managed to preserve consistency across the different aspects and guarantee a coherent development flow [?].

Industry has matured expertise in UML modelling which makes it a de-facto standard in system design and documentation activities. As a consequence, the adoption of domain-specific languages (DSL) to be developed from the ground up, while preferable from the viewpoints of quality of modelling, domain-specialisation, and precision, is simply not viable: typical industrial systems run for decades and require constant maintenance in a cost-effective way, which would be severely hampered by discontinued tool support, typical of not-standard solutions. Moreover, industry often develops and/or owns specialised tools and organisational arrangements which have grown entwined with the system [?].

The overall goal of CHESS (cf. Acknowledgments) was to enable fully automated code generation for complex systems with guaranteed preservation of extra-functional properties verified at model level. In this work we present the CHESS tool environment, from cross-domain modelling of industrial systems based on UML profiling and separation of concerns to model-based analysis and automated code generation both equipped with back-propagation features.

2. CHESS METHODOLOGY
The design of complex systems poses a multitude of challenges in both industrial and research settings, even with conflicts between them. In the following we discuss the way CHESS addressed some of those challenges.
2.1 Industrial Requirements

The elicitation of industrial requirements performed in CHESS (cf. Acknowledgements) resulted in the demand for: i) support for standard UML and its profiles; ii) user friendly modelling; iii) support for multiple design views; iv) compatibility with internal and proprietary tools and techniques; v) management of inter- and intra-model consistency; vi) adaptability to different development processes and standards. As it is often the case, even at this top level, conflicting requirements are evident: on the one hand, the modelling approach should be as domain-specific and tailored to the selected target implementation as possible, so as to enable efficient code generation; on the other hand, general-purpose modelling approaches are sought, possibly compatible with existing techniques and tools. In the same vein, the developed methodology shall permit the specification of the necessary details to facilitate specialised analysis of extra-functional properties, while at the same time the need is voiced for user-friendliness, simplified modelling, and lesser redundancy of concepts. Finally, a cross-domain modelling framework has to be realised to encompass multiple design views, while consistency across views and easy maintenance are required.

2.2 Methodological Approach

Separation of concerns, a long-known practice introduced by Dijkstra in [?], aims at keeping separation between the multiple and specialised aspects of software development. The central benefit of this practice is the ability to afford separate reasoning and focused specification. In CHESS we especially seek to attain separation between functional and extra-functional concerns.

IEEE 1471 [?] portrays the architecture of a system as organised into constituents called views. Each view is a partial representation of the system as seen from a specific viewpoint, that is the expression of specific concerns of a specific stakeholder. In CHESS we stipulated that each view should be the expression of a single (macro-)concern. In that manner, views become the key to the enforcement of separation of concerns in our approach to MDE.

In CHESS we went over and above IEEE 1471 by defining a modelling environment that allocates distinct concerns to distinct views. To make view-based development effective, it is important for all the supported views to be fully consistent with one another for each model element that may legally appear in multiple views. In general, two alternative strategies exist to meet this goal: 1) a synthetic approach, in which each view is modelled separately with one or more models which are later composed with the models resulting from the other views; 2) a projective approach in which the information that pertains to individual views is extracted from a single underlying model that describes the entire system. Both of them have advantages and drawbacks: the former allows the exploitation of domain-specific concepts carried by corresponding languages, at the price that consistency among views can be difficult to maintain, especially when manipulations entail semantic consequences that cannot be automatically derived [?]. The latter technique can prove less powerful in some cases due to the expressive power that is inherited from the underlying language, but has the great benefit of allowing consistency management by construction. In CHESS we chose option 2), as we consider inter-view consistency easier to attain than inter-model consistency.

3. CHESS TOOL ENVIRONMENT

The CHESS tool environment has been developed as a set of Eclipse plugins on top of MDT Papyrus (http://wiki.eclipse.org/MDT/Papyrus), an open-source UML design environment. More specifically, while the editing tools (e.g., editors, palettes) have been derived extending the native ones, the actual features (e.g., model-based analyses, code generation, back propagation, separation of concerns by design views) have been implemented from scratch as CHESS add-ons to Papyrus.

The CHESS modelling language (CHESS-ML) has been defined as a UML profile, including tailored subsets of SysML and MARTE profiles. Design views are defined as entities of the CHESS modelling language itself. As such, views are instantiated on each model as and when needed. The dire consequence of this contingent (yet currently inevitable) decision is that the concept of view is tightly coupled with the underlying organization of the model. This is unfortunate, and not easily portable. If views were first-level entities in the UML meta-model instead, we would immediately earn portability and important gains in time-and-space performance. Implementing a design view needs the encoding of: (i) its name and identifier and the enumeration of all the entities (UML elements, their associations and features) that it may contain together with editing rights; (ii) the rules which assure the well-formedness of the model and are to be enforced or checked by that view; and (iii) the rules that specify the conditions under which that view can be activated (e.g., dependencies with respect to other views).

Moreover, the controlled activation of design views can enforce a given flow of development activities: a development process in fact normally envisions multiple design phases to be carried out — some in potential parallelism, others in some specific ordering — quite possibly under the responsibility of distinct development actors. We regard design views as a very handy means to enforce the boundary between development phases. Therefore we made provisions for specific activation rules for design views.

The CHESS tool environment provides the following distinct views, each of which addresses a particular concern of the system under development.

Requirement View. It is used to model the software requirements and associate them to other model entities such as component implementations. The Requirement view is imported from SysML and the SysML Requirement Diagrams are used to support it. The user defines the software and hardware requirements (both for the functional and extra functional dimensions) in the CHESS Requirement View, with traces both to the upper level system requirements and to the modelling elements of the other CHESS ML views.

Component View. It is a set of views that represent the system’s break-down into components both from its functional and its extra-functional perspective. The component view is comprised of two overlapping views: Functional View: in this view the system is modelled according to component-based design, by which each component is equipped with provided and required interfaces realised via ports and with state-machines and other standard UML diagrams to express functional behaviour. Moreover, the Action Language for Foundational-UML (ALF, http://www.omg.org/spec/ALF/) is used to enrich the behavioural description; Extra-Functional View: it applies on top of the Functional
View. In conformance with the principle of separation of concerns adopted in CHESS, the functional models are decorated (as opposed to modified) with extra-functional information in this dedicated view thus ensuring that the definition of the functional entities is not altered.

**Deployment View.** In parallel with the software architecture definition, the CHESS development process requires the user to model the target execution platform. The Deployment view supports the modelling of the target execution platform and software to hardware components allocations.

**Analysis View.** It is a set of views in which the user can model the analysis contexts which will be used as input for some of the available analysis tools. This view is split in two distinct views, each specialised for a given type of analysis.

**Navigation among Views.** Fig. ?? depicts some of the main distinctive features representing the enforcement of separations of concerns in the CHESS tool environment. The distinct design views are automatically generated by the tool as UML packages in the Papyrus Model Explorer when the user creates a new CHESS model (A). Switching from a design view to another causes the change of a view indicator (set of different colours each of which representing a different view) placed on the toolbar. While switching among distinct views is done by navigating the model in the model explorer, the user switches from functional to extra-functional (and vice versa) by using the button placed on the left of the view indicator (B). Extra-functional definitions are modeled by means of decorations of already modeled functional entities (C). Switching among views affects also the model entities available in the customized CHESS palette; this enforces separation of concerns by driving the user in choosing among a set of entities (changing from view to view) in each of the different design phases (D).

**Figure 1: Separation of concerns distinctive features**

**Back-propagation and View Enforcement.** In order to provide a-priori guarantees of correctness, the results from analyses performed in the dedicated views (e.g., real-time and dependability) are automatically propagated back to the input model for the user to directly appreciate where refinements and corrections may be needed. A modelling view owns all its underlying model elements and diagrams. Read/write access rights have been defined for each view to enforce the separation of concerns by giving the write rights only to the owning view. However cross-view editing exceptions have been defined and an on-the-fly validator for views-related constraints constantly checks them.

In CHESS we faced the problem that not all of the constraints coming along with a given view can be checked while only listening for modifications at model level. Actions (e.g., drag and drop of entities between diagrams) performed through the modelling editor must be related with the graphical part, i.e. the diagrams, and the current view in order to be properly monitored. In CHESS each view is attached a given set of read/write permissions by the user, set per model entity. Owing to the lack of direct support for it from UML, this enhancement, which should have been added to the CHESS metamodel, had to live outside of it.

The on-the-fly validator is implemented as an extension of the EMP Model Transaction (http://www.eclipsae.org/modeling/emf/?project=transaction). A listener checks every user action on the model which triggers a set of model modifications packed in transactions. The listener identifies diagram and view the user is editing on thereby checking each modification against read/write permissions related to both view and diagram. If a permission or a constraint is not satisfied the entire transaction is rolled back and the user is notified. Moreover user actions can trigger automated actions which perform specific modifications to preserve the consistency of the model with respect to the modification introduced by the user. Cross-view editing exceptions are also implemented as read/write permissions. The permissions allow fine-grained control on how the model can be modified with respect to a given view and diagram; i.e., it is possible to define a permission on a single property of a stereotype for a given model element type with respect to a diagram associated to a design view.

### 3.1 Analysis Techniques

Separate design views are provided for two classes of analysis: (i) dependability (Dependability Analysis view) and (ii) predictability (RT Analysis view). Most importantly, for each technique have been implemented back-propagation features for enriching the design models with the analysis results thus enabling a multi-perspective extra-functional evaluation of the system.

**Dependability Analysis.** With state-based dependability analysis CHESS supports a quantitative evaluation of the dependability attributes of the system. The analysis is performed on a state-based stochastic model of the system [?]. Moreover, CHESS allows to automatically calculate the failure behaviour of an entire system from the failure behaviour of its components through the Failure Propagation Transformation Calculus (FPTC) technique [?].

Failure Mode Effects & Criticality (FMECA) [?] is provided by CHESS as a fault preventive method, since failure modes are identified early and their causes can be removed early in the development. CHESS also supports Failure Mode and Effect Analysis (FMEA) which is a bottom-up approach for the dependability assessment of a system where the propagation path for system failures is derived from initial errors which can be located in any subsystem [?]. While FMEA can apply a probability, consequence and detection scale calculating different failure modes and the corresponding failure rates using multiple simulation runs and statistics, FMECA is used to rate the criticality of a failure mode.

The top-down Fault Tree Analysis (FTA) [?] is offered in the CHESS tool environment as well. Such analysis mainly consists in the identification of lower level failure modes and analyses of the effect of multiple system level (functional) failures. Additionally, CHESS provides data-flow analysis at programming language level in order to statically verify that platform-specific implementations satisfy certain properties assumed by models.

**Predictability Analysis.** In CHESS the user model can be submitted to static analysis in the extra-functional dimensions of interest. For example, schedulability analysis verifies whether the timing requirements set on interfaces can be met [?]. The extraction of information from the user model (i.e., generation of a Platform-Specific Model, PSM, or a Schedulability Analysis Model, SAM) and generation of the input for the analysis tools are automated; the results of the analysis are propagated back to the design model as
read-only attributes of the appropriate design entities. The model transformation that generates the model representation for infrastructure code (described in section ??) in the SAM is an integral part of the analysis transformation chain: it consequently is able to add the semantic details necessary for the analysis to be as precise as possible. Thanks to the full automation, the analysis can be iterated at will until the designer is satisfied with the result.

3.2 Automated Code Generation

The CHESS component model prescribes separation of functional and extra-functional concerns which is naturally reflected in the produced code. For each specific domain an appropriate code generator has been developed from CHESS-ML to: Ada (space domain), Ada/C/C++ (railway domain), C++ (telecom domain), EAST-ADL2/AUTOSAR/-C/Java/RTSJ (automotive domain).

Generators primarily produce infrastructure code, which reflects the information available in the PSM. Except for the one developed for the telecom domain, the code generators do not produce code that implements functional behaviour. In fact, the general approach in CHESS is for code generators to treat the functional view separately from the rest. Functional code is intended to emanate from component implementations independent from extra-functional decorations. The infrastructure code addresses the extra-functional and deployment dimensions (thus covering, e.g., transparent distribution); it traps all invocations directed to and from components and operates the forwarding as required, transparently to the functional code.

The code generator for the telecom domain exploits the expressiveness of state machines and ALF to generate full-fledged functional behaviour. Consistency between design models and code is enforced by the generation of 100% of the code in an fully automated manner. Moreover, generated code can be analysed and results meaningfully propagated back to the source models for evaluating the preservation of extra-functional properties from models to code.

4. RELATED TOOLS

Several approaches and tool-suites have been proposed for the development of complex systems. Pride (http://www.idt.mdh.se/pride/) and Topcased (http://www.topcased.org/) are generic tool-suites integrating diverse analysis techniques; Topcased focuses on model checking while Pride integrates a wider range of analyses. Even if sharing the focus on extra-functional properties for generic systems, CHESS strengthens the notion of separation of concerns, the generation of 100% of functional code and introduces the further step of back-propagation of analysis results to source models. BridgePoint (http://www.mentor.com/products/sm/model\_development/bridgepoint/) is a commercial tool based on xtUML specialising on full code generation; extra-functional properties, as well as back propagation features are not exploited.

5. STATUS AND OUTLOOK

The CHESS project has come to an end on 30 April 2012. Its industry-acknowledged, flagship results include: (i) a solid, demonstrative implementation of separation of concerns by design views; (ii) a suite of model-to-model automated transformations for specialised system analyses with back propagation of results to the user space; (iii) fully automated model-to-code generation for both architectural and functional elements, with support for heterogeneous target languages. Each industrial partner involved in the project (e.g., aerospace, telecom, railways) has developed its own case-study and validated the features offered by CHESS. Plans for future work around the CHESS themes address a score of research and engineering issues, ranging from extending the modelling, verification and generation capabilities to multicore systems to allowing the support for design views to impact the evolution of UML and its derivatives.

6. ACKNOWLEDGMENTS

This research work was carried out within the CHESS project (Composition with Guarantees for High-integrity Embedded Software Components Assembly, ARTEMIS.JU grant nr. 216082, http://chess-project.ning.com).