Workshop proceedings of the 5th

Nordic Workshop on Model Driven Engineering

27-29 August 2007
Ronneby, Sweden

ISBN: 978-91-7295-985-9
Research report

Organizers

Ludwik Kuzniarz, BTH – general chair
Miroslaw Staron, IT University of Göteborg – program chair
Tarja Systä, Tampere University of Technology – special session chair
Mia Persson, BTH – local arrangement chair

Program committee

Magnus Antonsson, Ericsson, Sweden
Juergen Ebert, University of Koblenz, Germany
Klaus Marius Hansen, University of Aarhus, Denmark
Ebba Pora Hvannberg, University of Iceland, Iceland
Rogardt Heldal, Chalmers Institute of Technology, Sweden
Sune Jakobsson, Telenor, Norway
Kai Koskimies, Tampere University of Technology, Finland,
Johan Lilius, Åbo Akademi, Finland
Welf Lowe, Växjö University, Sweden
Jyrki Nummenmaa, University of Tampere, Finland
Ian Oliver, Nokia Research Center, Finland
Lars Pareto, IT University of Göteborg, Sweden
Ivan Porres, Åbo Akademi, Finland
Andreas Prinz, Agder University College, Norway
Claudio Riva, Nokia Research, Finland
Eleni Stroulia, University of Alberta, Canada
Tarja Systä, Tampere University of Technology, Finland
Sven Wenzel, University of Siegen, Germany
Jianli Xu, Nokia Research, Finland
Albert Zuendorf, University of Kassel, Germany
Kasper Østerbye, University Copenhagen, Denmark

Table of Contents

Preface

Modeling is an integral part of engineering disciplines and software engineering is not an exception. Models are built to both specify details for construction of software systems and to document software designs. The models are constructed using various methods and tools, such as the popular Unified Modeling Language, or emerging Domain Specific Modeling Languages. Modeling is intended to shrink the gap between the problem domain and the solution space by raising the levels of abstraction in software development and increasing the degree of automation. These, consequently, require new ways of developing the software, influencing such activities as requirements engineering, designing, testing, or running projects.

Nordic Workshop on Model Driven Engineering – NWMODE – aims to continue the traditions of the past series of Nordic workshops on UML (NWUML) held throughout the Nordic region. The intention of the workshop has always been to bring together researchers and practitioners working with modeling. The workshop topics include the emerging trends in modeling as well as empirical experiences of the existing methods.

This year's edition of the workshop contains a series of papers on the topics related to Software Product Lines, Domain Specific Modeling, model comparison techniques, or modeling of Service Oriented Architectures. It also contains two dedicated discussion sessions on the topics of quality in modeling and model comparison. The dedicated discussion sessions are intended to initiate discussions on the topics and hopefully originate collaboration between the participants.

As organizers of NWMODE 2007 we would like to thank all the contributors, authors and reviewers.

On behalf of the organizers

Miroslaw Staron
Program Chair

Keynote presentation

Quality in Modeling

Lars Pareto
Software Engineering and Management
IT University of Göteborg
lars.pareto@ituniv.se

In this talk, we will look into the meaning of *quality for designs models*. Many organizations that proceed from code centric to model based software development experience quality problems with their design models, and seek solutions to these. Numerous methods and tools to improve the quality of modeling exist, but do often, when evaluated by organizations, turn out to address some other aspects of quality than those in need for improvement. The purpose of this talk is to give a characterization of the quality landscape of design models (to guide improvement method selection and development). To this end, we introduce a *quality model* that encompasses qualities in need for improvement in real organizations as well as qualities for which well known improvement methods exist. The model is similar to ISO 9126 (a quality model for software products) but has partly different sub-characteristics; it is based on empirical research in two large organizations and on recent literature studies from within the modeling domain.

Keynote presentation

Architecture-driven software development

Claudio Riva
Nokia, Finland
claudio.riva@nokia.com

Model-driven engineering holds the promise of automating software development through the use of high-level models of the software. However, its industrial applicability is often hindered by the gap, often considerable, that exists between the high-level models of the software and the actual implementation. The architectural designs are often incomplete, sketched with informal notations and rarely conform to the actual implementation. In this talk, we investigate how to document the architecture of an existing system and how to ensure that certain architectural rules are enforced in the implementation. We will present a method and its formalism for linking the high-level models with the implementation. We will also present several case studies taken from the telecommunication domain.

On the Evolutionary Development of Service-Oriented Architectures

Eleni Stroulia

Department of Computing Science
University of Alberta
stroulia@cs.ualberta.ca

Abstract. The objective of this paper is to explore the relationship between the engineering of Service-Oriented Applications and some strategic and economic concerns of the organizations that (consider to) adopt this architecture style for the development of their software systems. To that end, (a) we discuss some pragmatic observations regarding the potential role of SOAs in the current economy, (b) we identify some distinct types of applications envisioned to be developed in the SOA style, (c) we correlate some strategic business decisions with different types of SOA evolution scenarios, and (d) we outline a novel model for estimating the ROI of such evolution scenarios. This work rests squarely within the newly articulated area of "Service Science, Management, and Engineering (SSME)" [15,19] as an "interdisciplinary approach to the study, design, and implementation of service systems, i.e., complex systems in which specific arrangements of people and technologies take actions that provide value for others".

1 The role of SOAs in a Services Economy

It is becoming increasingly apparent that the services sector accounts for a substantial percentage of the worldwide economic activity and that this percentage has been steadily increasing. According to IBM [25] "*The world is becoming one global service system. Five of the top ten nations ranked by labor force size have more than half of their labor working in services (versus agriculture or goods production). China and India are not yet in the top five – but with roughly 40% of the combined worldwide labor force, as these "sleeping giants" migrate towards services the implications are profound*." Even more importantly, the percentage of workers engaged by the services sector increases at the expense of the corresponding percentages of workers in agriculture and goods production. These statistics underline the increasing importance of the services sector in the world economy and emphasize the need for clarifying which exactly activities are included in this sector.

According to Zysman [26], there are four different types of economic activities that are accounted under the general label of "services".

1. The first is an artifact of financial engineering: activities outsourced from manufacturing are relabeled as services because they are conducted by different organizations, even when they remain essentially the same.
2. The second category involves an increasing number of the traditional activities

involved in business-consumer interactions and between-business interactions.
3. The third service category accounts for the conversion of unpaid domestic work (traditionally the responsibility of women) into commercial services, as women increasingly work outside the home.
4. The fourth category represents a fundamentally new and innovative class of service activities, facilitated by software systems, which are increasingly used to formalize, codify and drive the execution of business processes.

This last category, characterized as the "fourth service transformation" by Zysman, is the focus of this paper. Zysman points out that "*tools and technologies based on algorithmic decomposition of service processes may have the power to revolutionize business models, by displacing and by complementing the people (involved in these processes)*" thus implying a tight inter-dependence between the evolution of today's software systems and the business processes they support.

Service-oriented architecture is a new software-engineering paradigm, which is being increasingly adopted [39,40,41], and which advocates the design and development of complex software systems through the composition of (independently designed and developed) "services". The term "services" in this context refers to software components, which, although implemented in a variety of platforms and programming languages, are interoperable through open XML-based specifications of their interfaces.

A quite sophisticated stack of open standards (based on XML syntax) is currently under development to support the specification of various aspects of service-oriented applications. For example, in addition to the services' interfaces represented in WSDL [33], the run-time information exchange among services is governed by the SOAP protocol [31], and the composition of services (including data and control flow among them) is represented in BPEL [29]. These three specifications are quite mature, as evidenced by the variety of existing implementations and tool support. A set of less mature specifications are also under development to elaborate the nature of transactional support required for these complex systems with long-running workflows [38], the agreements between the collaborating services [36], and the relevant security requirements [37].

The use of the term "service" in these two contexts, namely as (a) a value-producing process between an organization and its customers (business context), and (b) a reusable software component delivering a coherent set of functionalities (software-engineering context), is an indication of the importance of the abstraction it captures: "software services" can be recursively composed to develop complex processes, whose execution, in turn, drives "business services". The declarative composition specification (in terms of WSDL and BPEL) constitutes an effective formalization and codification (as per Zysman's fourth definition) of the business processes.

The specification process changes, and in some cases it even revolutionizes, the current business processes since it explicitly considers which activities can and should be automated, which should be mediated by organization employees (acting in specific roles), which might be driven directly by the interaction between the customer and the SOA application, and which should be outsourced to SOAs of the organization's business partners (B2B interactions).

This phenomenon signifies a "phase transition" in the traditional interface between

business-process modeling and engineering and software engineering, the most significant aspects of which are as follows.

- Software-architecture descriptions are now visible to CEOs, who can consider specific investments in IT adoption and evolution in the context of their strategic business planning. On the opposite side, the economic and strategic business concerns can now more explicitly influence software-architecture decisions.
- We are increasingly witnessing the development of domain-specific repositories of software services, based on which complex compositions can be developed across individual organizations. Although the global UDDI [32] repositories envisioned by OASIS do not seem to gain sufficient momentum, business consortia develop their own standardized vocabularies [2,11] to describe their products, information and activities, thus paving the way to the development of shareable services automating these activities and the information exchange among them.
- The SOA stack of standards supports a rich set of metadata, about sector-specific vocabularies and software, which can be potentially considered in the context of business decisions.
  - Service-Language Agreements can be used to formalize different service variants, corresponding to different classes of workflows subject to different regulatory and performance constraints.
  - As application sectors define standard vocabularies and services, test-beds can be specified to define acceptable and desired performance targets, against which new service implementations can be measured.
  - Finally, as multiple services offering similar functionalities become available, their user programmability, as reported in developers' forums, becomes yet another criterion for their adoption.

2 Innovations in the Service-Oriented Computing Paradigm

Having reviewed the general economic environment to which SOAs are envisioned to contribute, let us now discuss the essential properties of this software design and development paradigm.

The SOA paradigm is not fundamentally new; in fact, the field has already attempted multiple times to develop standards for the composition of complex applications based on independently developed components available in shareable repositories. However, there is a set of important innovations that this new paradigm encapsulates, which makes its widespread adoption more likely.

Services are network-accessible components, which may have been developed in any of a broad range of programming languages. Their broad accessibility and utilization is enabled by their WSDL-specified interfaces and the fact that at run time they can be invoked through SOAP, an XML-based protocol. Never in the past have interface-description languages been agnostic of programming languages and platforms: CORBA has been partial to C++, COM/DCOM has depended on the Microsoft platform, and, more recently, we have seen a variety of Java-specific protocols (such as Jini for example) as part of "Java as middleware" trend [24]. The SOA standards, based on XML, are open in a way none of these precursors were and we are already seeing a variety of commercial and open-source IDEs supporting the

development of services in a variety of programming languages. Furthermore, SOAP, the run-time invocation is also XML based and can be implemented on HTTP (as well as on a variety of platform-specific protocols) which makes it less sensitive to firewalls and other organization boundaries. The choice of XML as the syntax underlying the SOA stack of standards constitutes an innovation of great potential importance for their widespread adoption.

The second important innovation is the flexibility of the service-discovery scenarios supported by the discovery-related standards. Originally, the UDDI registry structure and discovery protocol [32] envisioned globally accessible repositories of general services, organized under multiple overlapping taxonomies. However, there were not enough rich taxonomies to precisely classify organizations and their services, and no clear business motivation was articulated for why businesses should advertise their software services in these registries, which were too public and did not offer much support for the developers (beyond browsing). Consortia-specific and private UDDI registries are seen some adoption and the Web-Services Inspection Language (WSIL) [34] offers an alternative, lightweight scheme for the advertisement and discovery of services, which should be increasingly usable as specific sectors and consortia develop their own taxonomies for specifying the information they exchange and their interactions [2, 11].

Finally, the most interesting and powerful innovation of the new standards is the fact that they enable recursive composition of services in increasingly complex services specified in BPEL, whose interfaces are again specified in WSDL. Traditional distributed middleware has aimed at "transparency": through syntactic extensions to traditional programming-language constructs, the middleware designers have tried to hide from developers the complexities of accessing components outside their system boundaries. By making distributed components seem just like any other component – at the programming-language syntax level- the learning curve of the middleware frameworks is simplified, at the cost, however, of hiding the dependencies across domains of software ownership and control. This approach is eminently inapplicable to today's vision of enabling interoperation across organizations through the Internet. Different organizations want to integrate their software systems and processes in order to increase the effectiveness and efficiency of their partnerships. At the same time, they need to maintain control over how to manage their own software assets and how to grant access to their partners, subject to their own strategic goals and governance rules. To enable the flexible maintenance and possible evolution of processes involving multiple organizations, the rules of composition among components across organization boundaries have to be explicit. This is why BPEL is critically important for accomplishing the envisioned Internet-wide interoperability across programming-languages and organization boundaries.

3 Some Issues in the SOA Research Agenda

We believe that the software-engineering research agenda [14] for supporting the development and management of SOAs should cover the following four broad areas of research.

Defining a taxonomy of SOAs: The first concern is to identify the different types of SOAs, for which there is a pragmatic need. Not all new software systems should be developed in the SOA style and some existing systems should probably be migrated to an SOA style. The question then becomes to develop requirements-analysis methods that can recognize when an organization should migrate to an SOA [28] (whether through reengineering of existing systems or through development of new ones according to best practices associated with each distinct type).

Run-time SOA monitoring and adaptation: Once an SOA is developed and deployed, its run-time performance will have to be monitored to ensure that the orchestration instances are progressing as expected and the various SLAs applicable to the constituent services are respected. When orchestration instances fail or SLAs are (about to be) violated, autonomic, run-time reconfigurations might be possible to address the (potential) violations. Although there exist a variety of autonomic-adaptation algorithms for managing the performance of different types of computations (database query latency, network throughput, web-server request throughput) the task of end-to-end management, especially when the constituent components belong in different administrative domains, arises as a grand challenge in SOA research.

SOA evolution management: Inevitably, SOAs will have to evolve. Evolution is part of most software systems' lifecycles and it is bound to be even more important for SOAs, since evolvability is one of the main motivating factors for adopting the modular SOA style for an application. Given the modular nature of SOAs, the question then becomes to articulate (a) a taxonomy of "canonical" adaptations, (b) the opportunities that may motivate them in the operating environment of the organization that the SOA supports, and (c) the best practices for carrying out these adaptations. A critical element of such a "best practices" list is a method for understanding the economic trade-offs of SOA adaptations, especially when multiple alternative adaptations are possible, in terms of return-on-investment estimation.

Mixed service delivery with SOAs: Finally, we have to study the nature of interactions between SOA systems and users. This question will have to be informed by ergonomic and cognitive-science theories about what tasks people are good at (as opposed to tasks that people need help with) and by a study of the canonical roles that people (employees and customers) fulfill in service processes. The BPEL4People specification [30] is a first attempt at characterizing these roles, the nature of user-system interactions they imply and what the technical solutions for the necessary user interfaces might be. Clearly further study is required to refine our understanding of people-mediated and automated activities so that SOAs can be developed with an appropriate mix of the two types.

3.1 A taxonomy of SOAs

As an initial investigation of the question "what types of applications is an SOA good for" we have examined research publications and white papers to come up with a set of three distinct types of SOAs.

First, it is clear that SOAs, based on the web-services stack of standards, are perceived as the natural evolution of business frameworks that support complex business workflows. To date, we have seen organizations transition from legacy

systems, custom-built to the specifications of the organization, to (b) sector-specific frameworks – like SAP and PeopleSoft – customized to meet the differential requirements of each organization, to (c) J2EE (Java2 Enterprise Edition) frameworks that enable a greater degree of flexibility in system development, enabling the integration of COTS components with custom components, which sometimes may wrap legacy subsystems. This trend has been driven by the need to map the organization's processes onto its software systems (through integration of custom elements) while at the same time enabling (some) integration with partner systems (through reuse of general frameworks and integration of COTS). Both these requirements are extremely well met by the SOA standards, in fact better than all these previous middleware as application sectors standardize their information and interactions. Therefore, we expect to see a third wave of transitions to standard SOAs, especially from J2EE systems (since there is substantial tool support for this type of migration).

We believe however that there is an interesting conceptual difference between J2EE components and services, which should be accounted for in this transition. Although, services are frequently discussed as analogous to java interfaces and classes [9], they are meant to offer a coherent set of related functionalities, which together support an interesting complex capability and would be more appropriately conceived as use cases, for modeling and assembly purposes, which could be reverse engineered from the existing system, in cases of migration to SOA. Then, a WSDL specification can be developed to describe the external interface of the service and a BPEL specification can be developed to define its usage protocol. These two specifications can enable automatic means-ends compositions of services, possibly even at run time [5,6].

A different type of services is exemplified by mash-ups. According to Vint Cerf "mashups represent new ways of applying existing basic infrastructures, not originally intended by their designers" [1]. This ad-hoc and opportunistic development of services is becoming especially interesting when associated with an on-line social network, since it supports "long-tail" business activities [3]. Essentially, social-networking sites enable the formation of small niche markets of people with common interests, thus defining target markets for niche products and services, which might not be profitable in a more traditional economies-of-scale model. SOA as an enabler of efficient, low-cost service composition is well poised to become the de-facto architectural style for developing such niche applications.

The third distinct type of SOAs are motivated by the need to exploit cyberinfrastructure installations. The term cyberinfrastructure denotes a collection of powerful computational clusters, large-capacity storage devices, high-speed wired networks, ad-hoc, mobile networks and sensor networks. Today these installations are being used for special, resource intensive applications developed by researchers' teams, mostly in areas such as E-Science. The grid middleware [35] builds on the web-services standards to provide additional support for this type of applications, including job scheduling, data distribution and load balancing, aiming at the virtualization of the underlying resources and the simplification of application development on them, so as to encourage a wider variety of applications by a larger community. As the variety of grid applications and the types of underlying hardware resources increase, more middleware for enabling the efficient development of

applications needs to be developed. As grid services are increasingly adopted, we expect to see extensions of this type of SOA middleware, designed to enable the utilization of other types of resources, such as sensor and RFID networks. This is a necessary pre-requisite for exploiting the promise of these inexpensive devices that are envisioned to cover our world with "smart dust" feeding us with continuous information about it.

3.2 Run-time SOA Monitoring and Adaptation

Autonomic computing has been, in recent years, a fertile area of research, focusing (a) on predicting trends in the performance of the subject system and (b) configuring the system's operating parameters so that it meets its performance requirements. To our knowledge, most autonomic-computing solutions focus on monitoring and managing homogeneous system resources, at a small temporal granularity.

Two fundamentally new autonomic-configuration problems arise in the context of complex SOA applications. First, as a natural extension to autonomic performance management, research will have to address end-to-end management of complex systems. In this more complex conceptualization of the problem, one has to consider that the subject system utilizes a variety of resources and that, for any given anticipated violation, multiple alternative reconfigurations might be relevant, each one addressing different plausible root causes, having different impact to the system configuration, and implying different costs and risks. The objective then will be to synthesize multiple sources of performance metrics, to infer alternative root causes that could "cover" the collected evidence, and to decide on an economic and effective reconfiguration.

The second autonomic-management task is to recognize orchestration failures due to the independent evolution of the constituent services. Even when the WSDL specifications of these services do not change, their usage protocols may change due to the particular policies and regulations to which they may be subject. For example, suppose that Organization A starts imposing a particular ordering to the invocations of two of its operations that used to be independent. BPEL orchestrations that used to work will now start to fail. If BPEL orchestration middleware becomes explicitly aware of the usage protocols of the constituent services, when they evolve it will be able to automatically reconfigure the failing orchestrations to accommodate the changes.

An initial approach to addressing this problem is exemplified in our WRABBIT project [5,6]. This research examines the avoidance of orchestration failures that result from out-of-sync service-usage models. WRABBIT views service composition and coordination as a conversation among intelligent agents, each of which is responsible for delivering the services of a participating organization. Each such service is characterized by its WSDL interface and the abstract BPEL specifications of its usage protocols. In this context, an agent is a layer wrapping each peer organization, and is able to communicate with the other agents responsible for partner services, recognize mismatches between its own conversation model and the models of other agents (as these are revealed by conversation failures), and adapt the models as necessary to eliminate these errors. This approach applies to situations where the basic information and operations required for the successful completion of the agents'

conversation is available to them and failures can be addressed by renegotiating the conventions of the conversation. Further research is necessary to address failures resulting from actual differences between the required and available ontologies and operations.

3.3 SOA Evolution Management

3.3.1 SOA Modification Operators and Scenarios

Software-architecture research has recognized quite some time ago that modular design advances reuse, flexibility in adaptation, and ease of evolution [19,20]. When independent modules encapsulate distinct functionalities, making minimal assumptions about their operating environment, they can be reused in the context of multiple systems. Furthermore, systems designed as compositions of modules can evolve through local changes in their composition when the requirements in their behavior change. Thus, a modular design of a system encapsulates more "value" than a non-modular design of a system with similar functionalities.

Kazman and Bass explored this idea by demonstrating through case studies how various desirable non-functional qualities, such as scalability, separation, modifiability, integrability, portability, performance, reliability and reusability, can be supported by architectural design decisions. In their paper [13], they argued that these qualities can be achieved through the appropriate application of a set of "unit operations and that specific architectures that satisfy desired qualities can be derived through these unit operations. The unit operations proposed by Kazman and Bass were separation, abstraction, compression, uniform composition, replication and resource sharing. They were proposed only as a "candidate" set and they were derived by reflecting on software-development best practices.

Exploring this idea further and from a business perspective, Baldwin and Clark [4] examined specific software systems, like the IBM 360 for example, which owed their significant business success in their modularity. Based on their case-studies analysis, they formulated six modularity operators, in terms of which any change to a modular system can be described:

(a) Splitting - Modules can be separated into multiple independent modules.
(b) Substituting - Modules in a system can be substituted by other modules.
(c) Excluding - Existing Modules can be removed from a system to build a new solution.
(d) Augmenting - New Modules can be added to systems to create new solutions.
(e) Inverting - The hierarchical dependencies between modules can be rearranged.
(f) Porting - Modules can be applied to different contexts.

The Baldwin-and-Clark modular operators were proposed as a sufficient operator set for producing any desirable design. These operators are, to some degree, similar to the Kazman-and-Bass operations. Separation (i.e., the encapsulation of a distinct functionality in a module) can be accomplished by splitting a multifunctional module in multiple ones, each one dedicated to one of its original features. Abstraction (i.e.,

the specification of a virtual machine that can potentially be implemented by different interpreters) corresponds to the inversion operator; furthermore, it can be conceived as a mechanism for supporting portability. Compression does not really have a corresponding operator in the Baldwin-and-Clark set and neither does uniform composition and resource sharing. The replication operation is a restricted variant of the augmentation operator, where all new modules introduced are replicas of an existing one connected essentially "in parallel" to the original module.

Both these lines of work rely on similar methodological principles:

1) Modular system designs encapsulate value, manifested, in one case, as the accomplishment of desired non-functional qualities, and in the other, as economic gain.
2) Effective modular system design should be conceived as an evolutionary process; in fact, as both works demonstrate it frequently is such a process. The nature of this evolution appears to be systematic, depending on recurrent types of design structure changes, which the two operations' sets attempt to capture.

In this subsection, we espouse the same principles and we adopt the modular-operator set of Baldwin and Clark to discuss specific variants of these operators in the context of SOA evolution scenarios. In the following, for each modular operator, we identify potential business-strategy contexts that may motivate the application of the operator to the SOA system. Given the fundamentally modular nature of SOA applications, the objective of this discussion is to advance the case for a systematic, model-driven evolution of these applications, that relies on tool support relevant to the application of each operator and is documented by a value estimation of the impact vs. the cost of the operator's application.

Splitting: As a particular service becomes increasingly reused in the context of multiple SOAs, a variety of SLAs may be developed to further refine the constraints on its performance. For example, a given service may be invoked to provide just-in-time updates of a real-time variable, like stock prices, or a periodic report of the same variable at a lower cost to the subscriber.

Furthermore, different types of usage protocols may emerge as variants and/or refinements of its original usage protocol, based on the needs of its clients. For example, some clients may use only a subset of the possible combinations of the service operation parameters, i.e., for low-cost items in a product catalog users never exercise the additional insurance-buying option. According to the interface-segregation principle [16], clients should not depend on interfaces they do not use; therefore as particular variants of the original service become widely adopted, a new service should be developed to meet these specific needs.

To enable the recognition of service variants as candidates for becoming independent services, an automated monitoring capability could be exploited to recognize patterns in the SLA constraints and the usage record of a given service by its various clients. In addition, an automated workflow-refactoring process could potentially support the integration of the discovered variant services as potential alternatives for (some of) the roles that the original general service used to fulfill.

Substituting: Several different types of substitution scenarios exist. First, an organization may decide to provide an automated alternative for an employee-mediated task. For example, while previously a customer had to contact an employee to place an order, an on-line product-order form is now available. This type of

substitution does not appear to require any special support, if there already exists a user interface for the employee receiving the order; this user interface can be migrated to a web-accessible platform and can still drive the original order service.

Alternatively, an organization may decide to select an alternative partner for an outsourced service, such as instead of using the FedEx "Saturday delivery" service, they start using a similar UPS service. This scenario requires support for an example-based service discovery API: developers can use the interface and usage protocol of the service they currently employ as an example for discovering similar services that can be used to fulfill the role of the retired service. Neither UDDI [32] nor WSIL [34] offer such an API but a substantial body of research exists in this area, including some of our own integrating lexical, syntactic and semantic information available in WSDL specification to assess similarity [22,18]. This work is not mature enough to support automated discovery, which will eventually rely on semantic-web technologies.

Excluding: At any point in time, the process-monitoring engine may recognize that execution paths, although possible in terms of the BPEL-orchestration specification, are not actually exercised. For example, a path of a "pick" structured activity is never exercised, since users never choose to fill optional questionnaires. Alternatively, alternatively, some condition in a "case" structure activity is never met. In such cases, all redundant services may be excluded from the process, so as that the specification better reflects the actual practice. To support this type of modification, one envisions a BPEL-process transformation utility, which should be able to automatically transform the configuration of the affected BPEL structured activities, ensuring the consistency of the impacted data and control flow.

In cases where some parts of a service are used while others are not, when for example only a subset of the service API is used – when the monitoring engine for example observes that the high-cost stock-price service variant is not used - this modification may necessitate a "splitting" operator first, before excluding the not utilized parts of the service interface.

Augmenting: One can imagine a variety of augmentation scenarios, motivated by the organization's need to differentiate its offerings from those of its competitors and its own current offerings. The first is conceptually the inverse of the "splitting" scenario: through SLA negotiations, the organization may discover that an extended variant for an existing service is required. For example, in addition to the notification that an order is ready for pick-up, a feed for changes to the transit state of the order may be desired by some clients. Another augmentation scenario is conceptually the inverse of the "excluding" scenarios above: a new service may be added as one more option in a "pick" structured activity. For example, a new discounting mechanism is added to enable a new type of special pricing promotions. Both these types of scenarios require some support from the BPEL-process transformation utility.

The above augmentation scenarios have a rather local impact on the BPEL process specification. One can imagine a more challenging augmentation scenario. For example, the organization may decide to offer more features in its process, such as to enhance a product-catalog service with a recommender service. Alternatively, motivated by a strategic partnership, the organization may decide to integrate its processes with a partner's process. For example, in addition to selling the products in their own product catalog, a business may act as reseller to a partner product catalog, thus necessitating additional interactions with the partner accounting and warehousing

services. These types of augmentation modifications will require a sophisticated BPEL-process planning utility.

Inverting: Inversion scenarios are essentially standardization scenarios. For example, in order to enable the introduction of multiple alternative services where only one was invoked, the essential, and common across all alternatives, interface needs to be decided. Consider, for example, the case where a business product catalog enabled books' ordering only and has evolved to also enable ordering CDs. At this point, if the more general "product" concept is specified, any spurious book-related dependencies between the ordering process and the original product-catalog service can be eliminated.

This operation essentially addresses the intuition behind the dependency-inversion principle [17].

Porting: Two fundamentally different porting scenarios exist. The first involves the migration of an application from a traditional middleware platform to an SOA relying on the web-services tack of standards (e.g. from a J2EE framework to Apache Axis). This scenario can be supported by code-transformation tools. Although to our knowledge no such tools exist yet for this specific transformation, there is a substantial body of research in syntactic code transformations [10], which could provide a substantial basis for this task.

A different type of porting scenario may involve the migration of a service from one domain to another, such as for example, porting a book product-catalog service to the DVD domain. This type of modification would be motivated by the organization's need to expand its offerings and would require support with the alignment of the ontologies underlying the source and target service domains [27].

3.3.2 Return-on-Investment estimation of SOA decisions

We hope that the example scenarios motivating the SOA modification operators in the above subsection sufficiently demonstrate that the gap between investing in SOA development and evolution and enabling strategic business decisions becomes increasingly smaller. This implies the need of a more sophisticated model for estimating the cost and the value of any particular SOA evolution investment.

Traditionally, software-engineering economic models have focused on the prediction and analysis of the costs associated with the development of an application [7]. The implicit assumption is that the value of a software product is known from the outset, based on a contract between the software company and the business client. As the value is constant, profit for the software development company can be maximized simply by reducing operating and development costs.

In our recent work [23], we have been examining present an initial step towards such an integrated model for cost/value estimation of SOA evolution. We adopt COCOMOII [8] as a software-development cost-estimation model because it explicitly models software-development cost as a function of several factors of the reuse-based development process, which SOA represents. In effect, COCOMOII estimates the cost of creating a software-development project as a function of the complexity of the transformation of the application from an existing state to its envisioned state. In estimating the value of a software-development project, one has

to consider both the revenues produced directly by marketing the newly supported services and the non-tangible value of the potential service that could "easily" be built in the future using the evolved service-oriented application.

The flexibility and reusability of a service composition can be modeled with real option theory [12]. Real-option analysis, in conjunction with traditional service-valuation market research, enables the determination of the value of an SOA project. Combined with the cost analysis of developing and maintaining the system, a complete service-based economic model can be developed.

3.4 Mixed Service-Delivery with SOAs

Today, mixed-contact B2C service-delivery processes, i.e., processes incorporating automated, self-service and employee-mediated activities, are the de-facto standard for service delivery. For example, customers may order goods from the online product catalog of a business, pay through PayPal, have the products shipped to their address, receive them, and then benefit from ancillary services, such as an exchange policy, and a bricks and mortar store near them.

Given this range of interactions among consumers, employees and systems in the context of service delivery, the research question becomes to design and manage high-quality service-delivery environments, with multiple heterogeneous points of contact between the consumers and the environment. To date, SOA standards have been mostly silent regarding the roles that people play in the execution of BPEL-driven processes: any such interactions have to be hidden as asynchronous messages sent to the process. The BPEL4People specification [30] is layered on top of the BPEL language so that its features can be composed with the BPEL core features whenever needed. It represents an effort to define the types of roles that people play and the precise mechanism by which the user interfaces through which they interact with the SOA can be integrated with the core BPEL process.

In this context, the service-delivery environment should be designed to accommodate heterogeneity in how diverse groups of consumers utilize the mixed contact automated and employee-delivered service. In the end, the technical specifications of the automated and human service-service delivery systems must translate into customer-relevant service quality attributes (including service fees) that yield both customer satisfaction and organizational sustainability (i.e., profitability).

4 Concluding Remarks

In this paper, we have attempted to discuss a framework around research activities in the area of SOA engineering and to provide an overview of some important pragmatic concerns for this research. To summarize, we have tried to argue that

1) SOA is well suited to support traditional business processes, offering a huge potential inter-organization system integration. Furthermore, it is also appropriate for the development of ad-hoc, bottom-up integrations of existing services to deliver functionalities to niche markets of social networks. Finally, it can provide a powerful vehicle for software development for a wide variety of applications on cyberinfrastructure.

2) Run-time SOA management requires solutions to a more extended set of problems than the ones addressed by traditional autonomic-computing work, such as dynamic process workflow reconfiguration and end-to-end SLA management.
3) The evolution of SOA systems closely reflects the evolution of the organization business processes and it can be understood in terms of the Baldwin-and-Clark modularity operators. This type of systematic understanding also enables the cost-effectiveness analysis of each considered evolution scenario, based on software-development cost estimation and the real options created for the organization by the evolved SOA.
4) Finally, we argue that SOA orchestration should explicitly model the role of the users involved in their execution, as most organizations today rely on mixed-service delivery processes, involving multiple contact points among customers, employees and automated software systems.

References

1. Q&A: Vint Cerf on Google's challenges, aspirations http://www.computerworld.com/developmenttopics/development/story/0,10801,106535,00.html, November 25, 2005
2. Allianz Global Risks, Sekhon Associates: Allianz Global Risks: International Claims Reporting Using ACORD XML and Web Services. http://www.acord.org/Standards/pdfs/AllianzSekhon.pdf, February 2005
3. Chris Anderson: “The Long Tail: Why the Future of Business Is Selling Less of More”. Hyperion, 2006. http://longtail.typepad.com/the_long_tail/2005/01/definitions_fin.html
4. Carliss Y. Baldwin, Kim B. Clark: “Design Rules, Vol. 1: The Power of Modularity”, MIT Press Cambridge, MA, USA, 1999.
5. Warren Blanchet, Eleni Stroulia, Renée Elio: Supporting Adaptive Web-Service Orchestration with an Agent Conversation Framework. ICWS 2005: 541-549
6. Warren Blanchet, Renée Elio, Eleni Stroulia: Conversation Errors in Web Service Coordination: Run-time Detection and Repair. Web Intelligence 2005: 442-449
7. Barry W. Boehm, Kevin J. Sullivan: Software economics: a roadmap. ICSE - Future of SE Track 2000: 319-343
8. Barry W. Boehm, Alexander Egyed, Daniel Port, Archita Shah, Julie Kwan, Raymond J. Madachy: A Stakeholder Win-Win Approach to Software Engineering Education. Annals of Software Engineering 6: 295-321 (1998)
9. William R. Cook, Janel Barfield: Web Services versus Distributed Objects: A Case Study of Performance and Interface Design. ICWS 2006: 419-426
10. James R. Cordy: The TXL source transformation language. Science of Computer Programming 61(3): 190-210 (2006)
11. Philippe Deschênes: B2B Business Models- Case studies, MCETECH2006, http://www.michelleblanc.com/images/Presentation%20of%20various%20B2B%20Business%20models%20case%20study.pdf
12. M. Hakan Erdogmus, Jennifer Vandergraaf: Quantitative Approaches for Assessing the Value of COTS-Centric Development. IEEE METRICS 1999: 279-
13. Rick Kazman, Len Bass: Toward Deriving Software Architectures from Quality Attributes, Software Engineering Institute Technical Report CMU/SEI-94-TR-10.
14. Kostas Kontogiannis, Grace A. Lewis, Dennis B. Smith, Marin Litoiu, Hausi A. Müller, Stefan Schuster, and Eleni Stroulia, “The Landscape of Service_Oriented Systems: A Research Perspective,” Proceedings International Workshop on Systems Development

in SOA Environments (SDSOA 2007); Workshop at 29th IEEE/ACM Int. Conf. on Software Engineering (ICSE 2007), Minneapolis, Minnesota, USA; May 21, 2007, 6 pages, May 2007.
15. Paul P. Maglio, Savitha Srinivasan, Jeffrey T. Kreulen, Jim Spohrer: Service systems, service scientists, SSME, and innovation. Communications of the ACM 49(7): 81-85 (2006)
16. Robert Martin: "The Interface Segregation Principle", http://www.objectmentor.com/resources/articles/isp.pdf
17. Robert Martin: "The Dependency Inversion Principle", http://www.objectmentor.com/resources/articles/dip.pdf
18. Rimon Mikhaiel, Eleni Stroulia: Examining Usage Protocols for Service Discovery. ICSOC 2006: 496-502
19. Kevin Sullivan, William Griswold, Yuanfang Cai, and Ben Hallen: The structure and value of modularity in software design. In Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Vienna, Austria, September 10 - 14, 2001). ESEC/FSE-9. ACM Press, New York, NY, 99-108. DOI= http://doi.acm.org/10.1145/503209.503224
20. David Lorge Parnas: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12): 1053-1058 (1972)
21. Linda Dailey Paulson: Services Science: A New Field for Today's Economy. IEEE Computer 39(8): 18-21 (2006)
22. Eleni Stroulia, Yiqiao Wang: Structural and Semantic Matching for Assessing Web-service Similarity. International Journal of Cooperative Information Systems 14(4): 407-438 (2005)
23. Brendan Tansey, Eleni Stroulia: Valuating Software Service Development: Integrating COCOMO II and Real Options Theory, 29th International Conference on Software Engineering Workshops, 2007: 87-89.
24. Tom Welsh: Has Java become middleware? http://www.middlewarespectra.com/abstracts/2000_11_06.htm
25. John Zuk: "IBM Venture Capital Group Report" 19 Oct 2005, http://www-304.ibm.com/jct03004c/businesscenter/venturedevelopment/us/en/inthenewstemp/!!/gcl_xmlid=35968
26. John Zysman: The algorithmic revolution---the fourth service transformation. Communications of ACM 49(7): 48 (2006)
27. Ontology Alignment Evaluation Initiative, http://oaei.ontologymatching.org/
28. ISIS Approach: Service Migration and Reuse Technique (SMART) http://www.sei.cmu.edu/isis/smart.htm
29. BPEL – Business Process Execution Language: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel
30. BPEL4People – Business Process Execution Language for People: http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people
31. SOAP – Simple Object Access Protocol: http://www.w3.org/TR/soap
32. UDDI – Universal Description, Discovery and Integration: http://www.uddi.org/
33. WSDL – Web Services Definition Language: http://www.w3.org/TR/wsdl
34. WSIL – Web Services Inspection Language: http://www.ibm.com/developerworks/library/specification/ws-wsilspec/
35. OSGA – Open Grid Services Architecture, http://www.globus.org/ogsa/
36. Web Services Agreement Specification, http://www.ogf.org/Public_Comment_Docs/Documents/Oct-2005/WS-AgreementSpecificationDraft050920.pdf

37. Web Services Security, http://www.ibm.com/developerworks/library/specification/ws-secure/
38. Web Services Transactions specifications, http://www.ibm.com/developerworks/library/specification/ws-tx/
39. Web Services usage survey, CBDI report http://www.cbdiforum.com/bronze/webserv_usage/webserv_usage.php3
40. Adoption of Web Services and Technology Choices, TechMetrix Research http://www.techmetrix.com/products/products.php?type=rep
41. Gartner Surveys Show Web Services Are Entering the Mainstream http://www4.gartner.com/DisplayDocument?doc_cd=114570

A Feature-based Framework for Model-Driven Engineering

Susanne Busse[1], Helko Glathe[1], Thomas Stark[2], and Jianhong Zhang[1]

[1] University of Technology Berlin
Computation and Information Structures
Germany
{sbusse|hglathe|skipzr}@cs.tu-berlin.de
[2] Triggerware.de
thomas.stark@triggerware.de

Abstract. Model-Driven Engineering (MDE) improves software engineering by providing an appropriate level of abstraction for an early review of a system design. Model transformations can be used to establish a well-regulated development process. But existing transformation approaches still lack in connecting system requirements to the models realizing them. Furthermore, it is still difficult to integrate existing tools for particular tasks in MDE to an all-embracing environment.
This paper presents a framework for MDE by defining required components and data structures to establish such an environment. It combines concepts of a UML-based MDE with feature modelling from Product Line Engineering. Feature models capture characteristics of existing system components as well as requirements of a planned application. They are used to control all parts of the engineering process.

1 Introduction

With a component-based view on software systems, software development increasingly moves from a programming task to a component composition and integration task. In this context, two new paradigms of software engineering attract more and more interest: Model-Driven Engineering ([1, 2]) comes up with models and precisely defined model transformations that help to find an appropriate level of abstraction and are basis for analysis, system checks and simulations. Product Line Engineering ([3]) and Domain Engineering ([4]) allow to deal with system families that share a common software architecture. After modelling the family domain with its common and variant features (domain engineering), an application can be partly generated by generators (application engineering).

There already exist techniques and tools for specific tasks in Model-Driven Engineering but it is still difficult to integrate them into an all-embracing environment. Each tool has to be adapted to particular requirements, for example by adding domain-specific model languages, the models must be transformed between different metamodel versions, missing functionality must be realized, and an appropriate model management has to be established.

1.1 Objective

This paper presents a concept of a feature-based framework for Model-Driven Engineering (MDE) that defines required functionality of tools and main data structures. The main idea of our MDE approach is to combine a UML-based model transformation process with a feature-based variant management that connects features of applications with the models specifying how these features are realized. Thereby, a feature represents a characteristic of a system whose existence or absence can be noticed. We not only use feature models to characterize applications, bus also to characterize reusable knowledge for designing an application within the domain. So, we combine MDE and domain engineering concepts.

The development of our framework is mainly influenced by our experience in information integration systems. An Information Integration System (IIS) provides a single read-access point to a set of autonomous heterogeneous data sources ([5]). There exist a broad variety of IIS ranging from schema-based mediators to information retrieval approaches like search engines. So, IIS form a system family wherein the members differ both in functional and quality aspects, that are represented in a feature model.

The domain of IIS shows many typical characteristics to be considered when developing a component-based software system in general. Particularly, we always have to consider existing information systems that should be integrated into an IIS. So, the development always contains a reverse as well as a forward engineering step. In this paper, we will focus on the design of an IIS. Briefly described, the design consists of the following steps:

- Computational-Independent Design
  After analyzing both the systems to be integrated and the requirements on the integration, the architecture of an IIS is defined as an initial UML application model. Basis are reference architectures for particular types of IIS such as mediators or search engines. Note, that the architecture only defines the structure of IIS components and processes for management and information discovery. It does not contain any algorithms or implementations. They are added in the next step.
- Platform-Independent Design
  Now, the initial design is refined by selecting appropriate realizations for all parts of the IIS: we select data models, data mapping languages, add algorithms for query processing and so on. Thereby, we can use knowledge on when and how to use these techniques. We will describe them as a kind of patterns of the IIS domain model and provide them by the framework.
- Platform-Specific Design
  In a last design step an appropriate middleware technology is selected and the platform-independent model is translated into a platform-specific one. Again, we can use existing middleware technologies like J2EE or .NET and patterns describing how to use them.

The feature-based framework supports all of these engineering steps. Abstracting from the specific domain of IIS, it supports the following tasks:

- *analyzing* the structure, behavior and properties of existing (information) systems;
- *selecting* appropriate patterns from a domain model, for example a reference architecture, reusable components, appropriate algorithms, languages and technologies;
- *generating* refined application models that apply a selected pattern or integrate a reusable component into the application-specific design (reuse).
- *documenting* the application design including traceability between the models of different design steps.

The remaining paper briefly describes all parts of our framework. Examples will be taken from the domain of IIS. The structure is as follows: Section 2 gives an overview on the framework and motivates the use of feature models as the main concept for MDE. Section 3 to 5 then go into the details of analyzing existing systems, matching feature specifications to find appropriate patterns for an application, and generating refined application models by model transformations, respecitively. Section 6 discusses our approach and gives an outlook to future work.

1.2 Related Work

Our work is based on the new paradigm of Software Factories ([6]), Model-Driven Engineering ([2, 1]) and concepts of the OMG's Model Driven Architecture (MDA, [7]). According to these paradigms, models and model transformations form our engineering process and are basis of the framework. So far, we do not need domain-specific modelling languages as they are introduced in several MDE approaches ([8]) because information integration systems form a horizontal domain that does not depend on a particular application domain. But tools for domain-specific languages like [9, 10] may be integrated if they are based on a metamodel approach.

We extend the MDE approach using feature modelling known from Product Line Engineering and Domain Engineering ([3, 4]) to cope with variants. Feature models are used to specify features of applications (requirements) as well as features of patterns for the design of IIS. This allows to compare feature specifications to find appropriate patterns and to control the transformation process. Thus, we use parameterized model transformations to generate refined models that are consistent with given requirements.

Our framework contains patterns that can be selected and automatically applied to an application by specified model transformations. Basis for the definition are pattern descriptions ([11, 12]) that we formally specify and integrate into the domain model for IIS. The concept is similar to an aspect-oriented approach ([13]) that defines different aspects and composes these parts together. But we do not need explicit composition rules because we use the reference architecture as a frame. A model transformation rule, called integration rule, specifies how a pattern is applied by referencing this frame. So, the composition rules are already specified within the domain model.

2 Overview on the Framework

According to a domain engineering approach, the framework supports the management of a system family and reusable knowledge (domain engineering) as well as the development of particular systems of the family (application engineering). Figure 1 gives an overview on both parts that are described in the following.

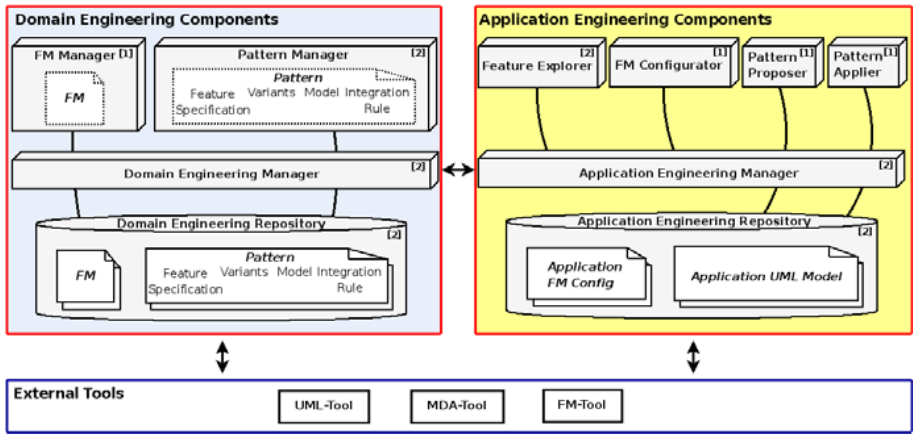


Fig. 1. Architecture of the Framework ([1] = implemented; [2] = designed)

2.1 Domain Engineering (based on the Framework)

The system family is described by a domain model that consists of

- general *feature models* that describe common and variable features of systems of the family. Thereby, features can represent functional as well as quality features.
- patterns defining reusable knowledge within the domain. 'Pattern' means here existing architectural or design patterns, languages, algorithms or even reference architectures of a domain - the granularity depends on the particular engineering phase. Each pattern consists of
  - a *pattern specification* that characterize the pattern according to the domain features described in the feature models mentioned before;
  - a (pattern-specific) *feature model* specifying possible variants of this pattern;
  - a *pattern integration rule* that defines how the pattern can be applied when developing a particular application and
  - a *pattern transformation rule* that configures the pattern according to the pattern-specific features when it is applied.

From a modelling point of view we assume a UML-based modelling when developing an application. Besides we use feature modelling ([4, 14]) to describe variants of systems and patterns. A feature model defines common and variable features of a concept, e.g. functional and quality features of an IIS. The feature diagram is a tree of features with the root representing the concept. The tree defines constraints on how these features can be combined when specifying a concept instance such as a particular IIS.

Figure 2 shows a small part of the feature model defining the system family of IIS on the left and a configuration on the right. When specifying a configuration, an engineer selects features he want to include in his IIS and removes features he do not want. Formally, this is a feature model configuration that defines a binding for each feature: Each feature can be bound, removed or undecided (neither bound nor removed).

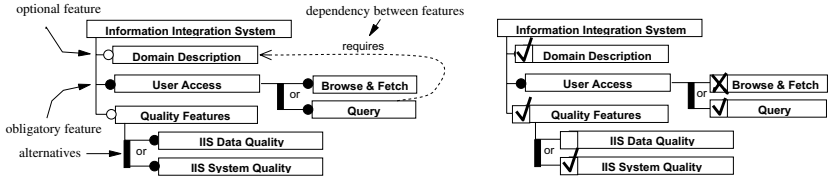


Fig. 2. Feature Models and Feature Model Configurations

The pattern transformation and integration rules are model transformation rules that transform UML models according to a particular feature configuration. They are an important part of the domain model as they extend the knowledge of a domain: the rules specify *how* particular features can be realized. So, they can be used to give a constructive support. Of course, model transformation rules are closely related to a feature model and thus domain-specific artefacts.

2.2 Domain Engineering Components of the Framework

The framework provides the following components to support a domain engineer to create or manipulate system family definitions.

FM Manager The FM Manager allows to create or manipulate definitions of feature models (FM). For this purpose existing feature modelling tools can be used such as pure::variants ([15]) or FeaturePlug-in ([16]).

Pattern Manager Patterns are defined using the Pattern Manager. It controls the definition of pattern specification, feature model, transformation and integration rules with their references to the general feature models.

Domain Engineering Repository All information of a domain model is stored in the Domain Engineering Repository (DER). It is based on a metamodel defining the main elements of feature models, feature configurations, UML models, rules and the relationships between them.

Domain Engineering Manager The Domain Engineering Manager (DEM) provides an interface to the DER and manages the domain engineering part of the framework. Thus, each component that is integrated into the framework must be able to communicate with the DEM.

2.3 Application Engineering (based on the Framework)

The application engineering part of the framework supports application engineers developing particular systems that are instances of the system family described with the domain model. Besides the *UML application models* already known from software engineering without any domain engineering, we now additionally manage *application feature configurations* that describe the particular application with respect to the system family.

Using the framework, an application engineer develops a system based on the already defined system family. The feature models of the DER allows to define requirements of the application. For each development phase the frameworks offers patterns that could be useful. The applications engineer can choose patterns, configure them according to particular requirements and then apply them using the defined rules of the patterns. The result of applying a pattern is a refined UML application model that integrates the pattern into the existing application model. So, the engineering process consists of several steps of applying patterns that continuously refine the system design.

The process defined so far only considers the development of a new application. But the framework also supports application engineers when analyzing existing systems. It provides tools to determine UML models and features of existing systems in order to enable interoperability.

2.4 Application Engineering Components

The following components support the application engineering:

Feature Explorer The Feature Explorer supports reverse engineering, i.e. it supports the automatic identification of features of existing systems. The result are feature configurations according to a set of feature models from the domain model. The feature explorer is described in Section 3.

FM Configurator Configurations of feature models can be constructed manually using the FM Configurator. An application engineer takes a desired feature model and makes a choice for optional features. For this purpose we again use existing tools for feature modelling.

Pattern Proposer This component identifies appropriate patterns for refining the application design. Basis are the feature configuration for the application and the pattern specifications. A matching algorithm – one way for identifying patterns – is discussed in Section 4.

Pattern Applier The Pattern Applier refines an application model by integrating a selected pattern based on the specified integration rule of the pattern. Basis is a model transformation approach (see Section 5).

Application Engineering Repository (AER) Similarly to the DER, the Application Engineering Repository stores application UML models and application configurations. Of course, it refers to the DER in order to preserve references to the domain model.

Application Engineering Manager (AEM) The Application Engineering Manager manages the AER. It is closely related to the DEM because there exist many relationships to the DER. Again, all components supporting the application engineering have to communicate with the AEM.

3 Discovering Features by Reverse Engineering

Beside the possibility to define features and requirements of an existing system from scratch using the FM Configurator (Fig 1), our framework provides a Feature Explorer component to extract feature configurations of these systems by reverse engineering. Reverse engineering of features has many applications. In this paper we will focus on identifying features of IIS data sources – needed to design an integration layer upon them.

3.1 Observing a Running System

There are many ways to generate code from abstract models, but there exist only a few frameworks working the other way round. Due to the arbitrary intricacy of code and missing information about the path coverage or what are typical input values makes feature exploration from source code a complex task. Therefore we rather observe a running system at work than assuming features from a static code analysis ([17, 18]).

A prerequisite for this approach is an application independent interface to gather runtime events. Therefore we use the debug interface at first, but we have also designed a class loader, which enhance the ordinary byte code at runtime with additional call events that log relevant information in an internal database.

Our approach to identify features from an existing application consists of three phases shown in Figure 3:

- During the *Analysis Phase*, we observe a running application at work. In this phase we gather static runtime informations about code features, which can of course also extracted from source code. But additionally we get information about typical paths through the code, its input and output values as well as statistical information about quality features like throughput or latency. Due to complexity can this additional information hardly be extracted by static code analysis. Our observing approach implies that the analysis phase have to be as long as all relevant use cases walk through for at least one time and that the pool of available information is as good as the observed system works in a representative mode.
- During the *Enrichment Phase*, we enrich the collected information with meta data, e.g. tagging of architectural and software pattern ([11, 12]), compute

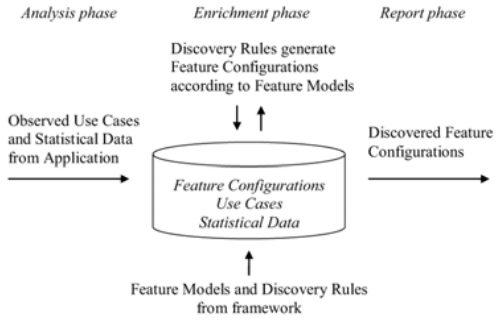


Fig. 3. Identify Features from Existing Applications

statistics about input and output values and identify features that are aligned with feature configurations.
Accordingly there exist pattern-oriented discovery rules with knowledge about typical patterns and how to find them in an application as well as feature-oriented discovery rules to identify features.

– During the *Report Phase* the feature configurations are built and returned to the framework.

3.2 From Runtime Information to Feature Configurations

The main task of the Feature Explorer is to transform the the raw runtime information to feature configurations. Therefore it gets a couple of discovery rules that are aligned with one ore more features. Applying a rule means to detect the existence or absence of a feature, which is remarked as the binding in the feature configuration: the feature is bound or removed. Of course, it is also possible that we can make no decision about it so that the feature will remain undecided in the configuration. If two or more rules that are aligned with the same feature came to different results, we use a voting algorithm to give a probability.

Although each discovery rule contains an autonomous algorithm, that works independently from all other, every rule cannot only access the observed information from analysis phase but also the generated information from previous working rules. The first discovery rules generally just format the information and identify common features while the secondary discovery rules take this information to identify more complex features. The basic rules are mostly domain independent whereas the complex rule algorithms are trained for IIS domain, but we also designed a plug in interface to easily extend the set of available rules and to adapt the feature explorer on other domains.

In the context of IIS, typical feature domains of interest are

Data Structure means that we made assumptions, if the data is
– structured, with atomic attributes or a fixed input vocabulary

– semi-structured, like RDF or XML streams
– unstructured or with an unknown structure, i.e. byte-streams

With long term observations it is even possible to make assumptions about the input and output scheme.

Information Discovery Method means that we analyze the structure and restrictions of interfaces for accessing information, as shown in Figure 8.

Summarizing, our approach has two main advantages: Firstly the feature discovery is based on analyzing both code-driven aspects and input-driven runtime information. Secondly it addresses a realistic situation wherein model-level and code-level do not evolve consistently. Here, our approach enables application engineers to synchronize these levels again.

4 Select Patterns by Feature Configuration Matching

The Pattern Proposer of the framework supports application engineers to find patterns, described in the DER, that can be used to realize particular aspects of their application. Thereby, the framework provides different methods to explore or search for appropriate patterns. In this paper we describe how feature configuration matching can be used for a pattern search. It is the most general approach: It similarly takes into account all aspects specified within feature configurations. And it not only allows to find appropriate patterns but also to rank them with respect to the feature configuration of an application.

A detailed description of the Eclipse plug-in for feature configuration matching can be found in [19] and a larger example on the use for IIS in [20]. In the following we call the component *FM Configuration Matcher* that is a part of the Pattern Proposer.

4.1 Matching Feature Configurations

Figure 4 shows the matching process examplified by the selection of an appropriate reference architecture for an IIS application. Basis for the matching is a feature model defining the variants of IIS. In domain engineering, this feature model is used to specify the features of existing reference architectures (step 1). We call such feature specifications *feature model specializations* because they do not fully specify particular systems but still families of systems - a subset of all IIS like mediator-based IIS or search engines.

On the other hand, an application engineer uses the feature model to specify the features of a particular application (step 2 to 4). The FM Configuration Matcher then matches this configuration with all the IIS specializations from the DE repository (step 5). The application engineer gets a quantified match result which tells him the best suited IIS type and a detailed result about conflicts and suggestions for each feature.

From a technical point of view we use feature models and staged configurations known from feature modelling ([14]) to express both specializations and

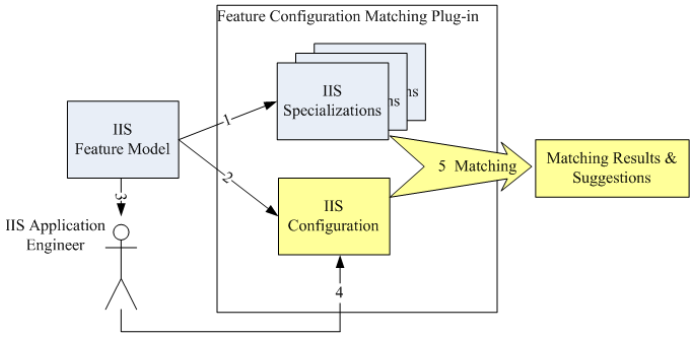


Fig. 4. Feature Matching Procedure

configurations. A feature specification, e.g. for a particular reference architecture, is built the same way as a configuration: A domain engineer decides which features must be present in an IIS of this type (bound) and which features do not occur in any IIS of this type (removed).

4.2 Feature Configuration Matching Algorithm

As both feature models and configurations has a tree structure, the matching algorithm consists of two parts: a comparison of the binding of each feature node leading to a detailed analysis of conflicts, and a traversal of the tree calculating a similarity for the whole feature tree. Thereby, the similarity of children nodes influences the similarity of father nodes. By iterating, we can get the similarity of the root nodes of configuration and specialization. The higher the similarity is, the more suitable is the reference architecture for the IIS application.

Table 1 shows the similarity values when comparing the two bindings of a feature node F - the result of a function *sim(F)*. The result is a value between 0 and 1. Besides this quantified values we have also defined conflict classes to classify the comparison result.

Spec. / Config.	BOUND	REMOVED	UNDECIDED
BOUND	1	0	0.5
UNDECIDED	0.5	0.5	0.5
REMOVED	0	1	0.5

Table 1. Feature Similarity Values

Based on this, the similarity of a feature tree is calculated as follows:

$$treeSim(F) = \begin{cases} sim(F) & \text{if } leaf(F) \\ \sum_{i=1}^{n}(treeSim(c_i))/n & \text{if } featureGroup(F) \text{ with children } c_i \\ sim(F)\sum_{i=1}^{n}(treeSim(c_i))/n & \text{if } feature(F) \text{ with children } c_i \end{cases}$$

4.3 Example

As an example we show the matching of a small part of a configuration for an IIS and the specialization that characterize the reference architecture of mediator-based systems (MBIS) (Figure 5).

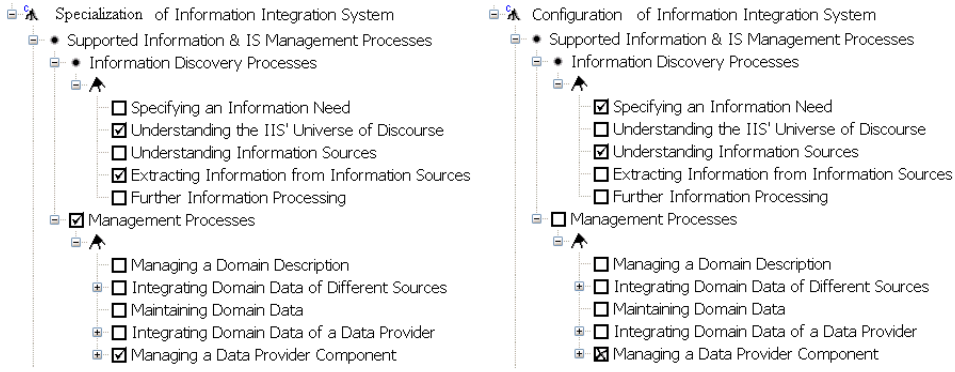


Fig. 5. Specialization of an MBIS System and Configuration of an IIS System

Supported Information & IS Management Processes	BOUND	BOUND	0.32166666
Information Discovery Processes	BOUND	BOUND	0.5
reference1IXIgroup0/reference1IXIgroup0			0.5
Extracting Information from Information Sources	UNDECIDED	BOUND	0.5
Further Information Processing	UNDECIDED	UNDECIDED	0.5
Specifying an Information Need	BOUND	UNDECIDED	0.5
Understanding Information Sources	BOUND	UNDECIDED	0.5
Understanding the IIS' Universe of Discourse	UNDECIDED	BOUND	0.5
Management Processes	UNDECIDED	BOUND	0.14333335
reference2IXIgroup1/reference2IXIgroup1			0.2866667
Integrating Domain Data of a Data Provider	UNDECIDED	UNDECIDED	0.20833333
Integrating Domain Data of Different Sources	UNDECIDED	UNDECIDED	0.225
Maintaining Domain Data	UNDECIDED	UNDECIDED	0.5
Managing a Data Provider Component	REMOVED	BOUND	0.0
Managing a Domain Description	UNDECIDED	UNDECIDED	0.5

Fig. 6. The result of feature matching

Figure 6 shows the matching result. There only exist one really conflict: the management of data provider components required in an MBIS but removed in the requirements specification. All other features show only potential conflicts because the feature is not bound or removed in one of the given specifications.

As an example, the similarity of 'Management Processes' is 0.14333335. The average value of all the sub-feature similarities is: (0.20833333+0.225+0.5+0.0+0.5) / 5 = 0.2866667. As the feature group node itself has a similarity of 0.5, the similarity of "Management Processes" is 0.2866667 * 0.5 = 0.14333335.

5 Applying Patterns with Model Transformations

5.1 Parameterized Model Transformation

The engineering process not only consists of analyzing existing information systems and matching features of components but also contains a forward engineering part. Here, we refine application design models and finally derive source code. In MDE, this refinement is described by a model transformation process.

Again, we can use feature models to control this process: Feature models describe variants we have when refining a model. Thus, model transformation rules should be parameterized by these feature models so that the transformation can be adapted to a particular situation specified by a feature configuration.

Figure 7 shows the general use. Starting point is an application model (Step 1). For its refinement, the developer selects an appropriate pattern from the domain repository that should be applied (Step 2). After selecting desired features of this pattern (Step 3), a model transformation rule generates the refined application model that now includes the applied pattern (Step 4). In [21] for example, you can find model transformation rules that generate platform-specific EJB models from a platform-indepent component model. There, the feature model specifies the criteria that are relevant to select an appropriate realization of a system with EJB components.

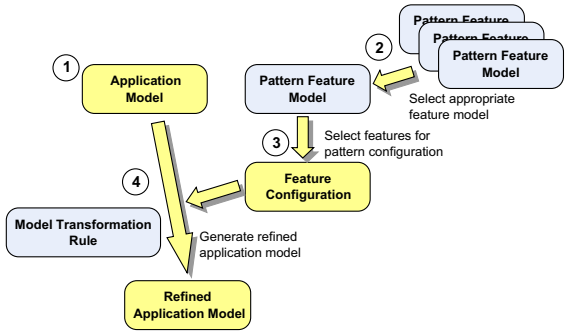


Fig. 7. Using Parameterized Model Transformations

5.2 Specifying Transformation Rules

To specify and execute model transformation rules, there already exist different languages, a good classification can be found in [22]. When selecting a language for MDE the following aspects should be kept in mind:

– Transformation rules cause a different amount of change ranging from adapting stereotypes to the generation of a whole UML model in a first design step.

It should be easy to specify transformation rules of both types. Therefore we support both small transformation rules directly specified with a transformation language and a graphical, generic specification using a UML extension. Then, a UML model is specified that represents the target model wherein particular elements can be annotated with constraints on the corresponding feature model. We call such a specification a UML transformation model.
- The amount of changes also influences the desired transformation semantics: Small changes should be done in-place whereas in other cases the construction of the target model is more convenient.
- A transformation language should be able to take several models as its input because we specifiy transformation rules that take a UML model and a feature configuration to build the target UML model according to the context given by a feature configuration. Again, only the first design step differs from this structure: These transformations really generate a UML model by only taking a feature configuration.
- The transformation language must cope with models of different metamodels as we use feature models and UML models. In later design steps one may introduce further metamodels of data models or code artefacts.

In [21] we use TRIPLE for specifying transformation rules. Now, we make experiments with ATL ([23]) because it looks set to be flexible enough for all the requirements described above.

Example

Again, we consider the design of an IIS. Figure 8 shows examples from the computational-independent design. Here a first architecture is generated according to functional features selected by an application engineer.

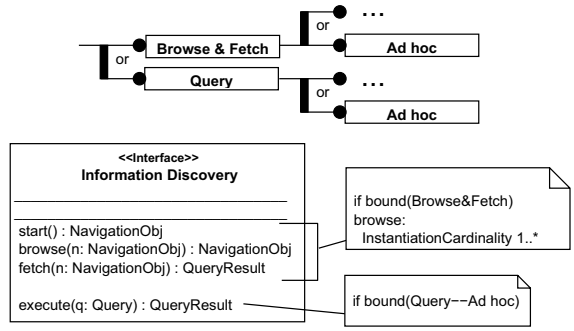


Fig. 8. Parameterized Model Transformations – Example

The figure specifies two variants of a discovery method of an IIS. The feature specifications on the top ('Browse&Fetch' or 'Query') are features of the corre-

sponding feature model of IIS. The variant UML model elements are annotated with a constraint on this feature model. In addition, an initialization cardinality can be specified such as for the browse method. This indicates that an application engineer can define one or more methods to browse through a navigation structure. So, the UML result model also contains hints for application engineers how to adapt a generated model.

6 Conclusions

This paper presented the concept of a framework for Model-Driven Engineering that extends existing UML-based MDE approaches by feature modelling so that we can deal with variants in software engineering:

- Feature models define common and variant properties of a domain and specify dependencies between them. This way, they give a structured definition of important aspects that have to be clarified in a particular engineering step or when reusing a particular component or pattern.
- Feature models allow to characterize both requirements on a planned system and properties of reusable patterns identified for the domain. Based on such specifications a matching can be used to find appropriate patterns and to identify conflicts when combining them.
- Feature models allow to define parameterized model transformations that generate solutions according to particular situations or requirements. This way, feature models fill the gap between requirements and solution supporting traceability and understanding of system models.

According to a domain engineering approach, the framework strictly distinguishes a domain engineering part managing reusable patterns and its specifications, and an application engineering part that supports the engineering of particular instances of a system family.

The framework mainly deals with UML and feature models (and their metamodels), feature configurations, discovery rules and model transformation rules that are part of the domain model and pattern specifications. These elements are basis for supporting both a reverse engineering that discovers features of existing system components and a forward engineering including the selection of appropriate patterns, their integration and the refinement of application models by model transformations. Thereby, we follow a component-based design of the framework that allows to plug-in existing tools for particular tasks in MDE. Future work will focus on supplementing our framework and refining the repository structure, mainly influenced by further experiments in the field of information integration systems.

References

1. Mellor, S.J., Clark, A.N., Futagami, T.: Guest editors' introduction: Model-driven development. IEEE Softw. **20**(5) (2003) 14–18

2. Schmidt, D.C.: Model-driven engineering. Computer **39**(2) (2006) 25
3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Kluwer (2001)
4. Czarnecki, K., Eisenecker, U.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley (2000)
5. Leser, U., Naumann, F.: Informationsintegration. dpunkt Verlag (2006)
6. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. John Wiley & Sons (2004)
7. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, Object Management Group (OMG), Architecture Board ORMSC (2001)
8. Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., Neema, S.: Developing applications using model-driven design environments. Computer **39**(2) (2006) 33
9. MetaCase: MetaEdit+. http://www.metacase.com/mep/ (2007)
10. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the use of graph transformation in the formal specification of model interpreters. Journal of Universal Computer Science **9**(11) (2003) 1296–1321
11. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
13. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proc. European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag (1997) 220–242
14. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1) (2005) 7–29
15. pure systems: pure::variants. http://www.pure-systems.com/ (2007)
16. Antkiewicz, M., Czarnecki, K.: Featureplugin: feature modeling plug-in for eclipse. In: Proc. eclipse '04: OOPSLA workshop on eclipse technology eXchange, ACM Press (2004) 67–72
17. Merdes, M., Dorsch, D.: Experiences with the development of a reverse engineering tool for uml sequence diagrams: a case study in modern java development. In: PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java, New York, NY, USA, ACM Press (2006) 125–134
18. Briand, L.C., Labiche, Y., Miao, Y.: Towards the reverse engineering of uml sequence diagrams. In: WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (2003) 57
19. Zhang, J.: Development of a tool environment as eclipse plug-in for model driven engineering of information integration systems. Master's thesis, TU Berlin, Fakultät IV Elektrotechnik und Informatik (2007)
20. Busse, S., Freytag, J.C.: Entwurf von Informationsintegrationssystemen auf der Basis der Merkmalsmodellierung. In: Datenbanksysteme in Business, Technologie und Web (BTW 2007). Volume 103 of LNI., GI (2007) 192–211
21. Billig, A., Busse, S., Leicher, A., Süß, J.G.: Platform Independent Model Transformation based on TRIPLE. In: Proc. Middleware '04, Springer (2004) 493–511
22. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3) (2006) 621–645
23. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005. (2005)

A Review of UML Model Comparison Approaches

Petri Selonen

Institute of Software Systems, Tampere University of Technology,
P.O. Box 553, FI-33100 Tampere, Finland
petri.selonen@tut.fi

Abstract. Several approaches for comparing UML models have emerged in the academia. They vary in detail like the way they identify corresponding model elements, subset of UML they support, input and output formats they support, and assumptions they make about the models. These approaches mainly focus on detecting and reporting differences between models, typically consecutive versions of UML class diagrams. This review paper presents a synthesis of the key characteristics of currently available UML-based model comparison approaches and compares their commonalities and differences. In addition, it discusses some of their potential usage scenarios and puts forward a call for their more systematic evaluation.

Keywords: UML, model comparison, model differencing, model merging, correspondence

1 Introduction

Software engineering is an iterative and collaborative process, producing series of related models describing the designed system from different viewpoints, at different levels of abstraction, or at different phases of evolution. Reasoning about the commonalities and differences among models is a fundamental activity to promote understanding of the system as a whole. As the number and size of models grow, maintaining a clear picture of the system becomes hard without proper tool support.

It is well established that conventional version control systems and text-based differencing tools fall short on reasoning about models with logical structure such as Unified Modeling Language [6] diagrams (see e.g. Xing and Stroulia [13], Ohst et al [8], Kelter et al [3]). Representing the models with hierarchical structures like XML has problems as well: XML differencing tools are agnostic of the UML semantics and do not generally match the developer's intuition very well. To address these shortcomings, several model comparison approaches supporting UML have emerged in the academia. They vary in detail like the way they identify corresponding model elements, subset of a UML they support, the input and output formats they support, and assumptions they make about the models.

This review paper presents a survey on existing model comparison methodologies and accompanying tools. The goal is to present a comprehensive coverage of the current body of knowledge in the area of UML based model comparison. The paper

presents a summary of selected model comparison approaches, synthesizes their key characteristics, and compares their commonalities and differences. Based on the review, a set of desirable qualities for the approaches, as well as a set of usage scenarios, are laid out for discussion.

The paper is organized as follows. Section 2 gives a short summary of the selected approaches. Section 3 presents a synthesis and comparison of the approaches. Sections 4 and 5 suggest some desirable qualities and usage scenarios for the approaches, respectively. Finally, some concluding remarks are given in section 6.

2 Summary of selected model comparison approaches

Five model comparison approaches have been selected to be covered in this paper. Before summarizing the approaches, a few definitions are given. Elements in different diagrams which provide a design for the same concept are said to *correspond*; *corresponding* model elements are model elements that represent the same semantic concept (Selonen and Kettunen [12]). A *model comparison approach* comprises both a methodology and an accompanying tool for comparing two models against each other. Figure 1 illustrates comparing two UML class diagrams and presenting their commonalities and differences in a unified diagram using colors.

In what follows, five selected model comparison approaches are presented. For each approach, a short description is given together with some of their key characteristics. While the author has made an effort to select the relevant related work, the list is not exhaustive. The presentation is based on the referenced publications only.

Girschick [2] presents an approach for automatically detecting and visualizing differences between UML class diagrams. The correspondence derivation is based on a set of class diagram specific evaluation functions, each presenting certain model element specific heuristics for evaluating the similarity of the compared elements. The result of the differencing is presented to the user as a unified class diagram with colors, and as an HTML table.

Kelter et al [3] present a generic difference algorithm for calculating differences between UML models. The correspondence is derived using a weighted similarity measure between elements according to criteria defined in a configuration file. The paper presents examples with UML class diagrams, but also claims support for statechart diagrams. As with Girshick, the result of the differencing is presented to the user as a unified class diagram with colors.

Ohst et al [8] present an approach[1] for visualizing differences between UML diagram versions stored in software configuration management systems. The visualizer tool is built on top of a version control system and supports class diagrams. It presents the common and specific parts of compared diagrams with the specific parts highlighted.

[1] The work of Ohst et al can be seen as a precursor of the work by Kelter et al.

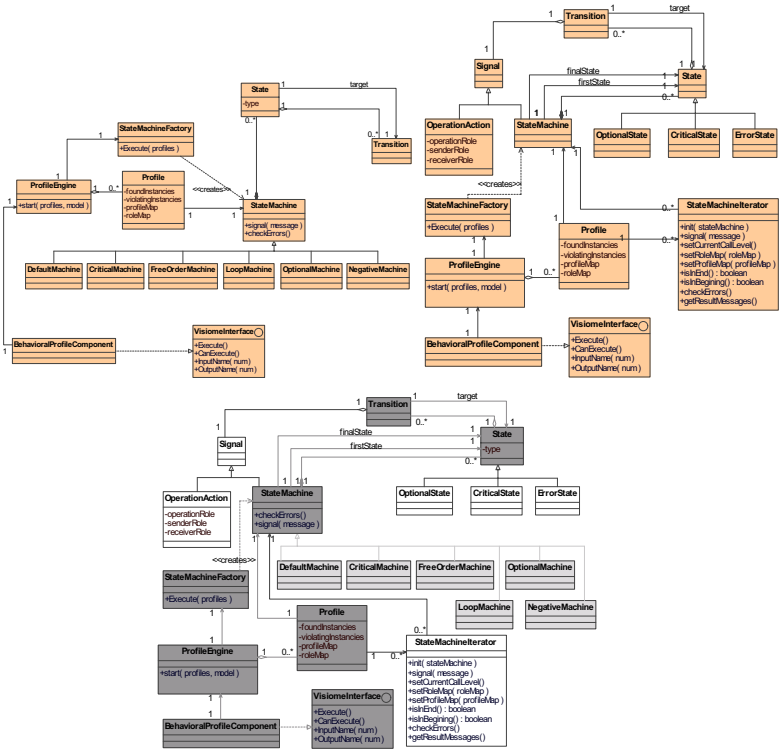


Fig 1. Example of comparing two UML class diagrams (adopted from [12]).

Selonen and Kettunen [12] present an approach[2] for generating correspondence derivation rules based on the abstract syntax of a modeling language, given as a MOF [7] metamodel. The correspondence is used as a basis for *UML set operations* (i.e. union, intersection and difference) resembling their mathematical counterparts. The authors give examples on applying the operations on comparing class and activity diagrams on both UML 1.5 and 2.0. The results of the operations are presented to the user as Rational Rose diagrams with colors, and in textual format.

Xing and Stroulia [13] present *UMLDiff*, a UML-aware algorithm for automated structural differencing of UML class models. The approach assumes that the class models originate from subsequent versions of object-oriented software, namely Java code. The algorithm identifies corresponding model elements based on the similarity of their names and structure. The tool presents the results to the user as a tree of structural changes in an Eclipse tree view. The approach concentrates on observing the evolution of a software system on a design level.

[2] The original and the metamodel-based approaches have been reported by the author of this paper in Selonen [10] and Selonen [11].

Several commercial tools offer simple, proprietary model comparison and merge capabilities. These tools include (in no particular order) MagicDraw, IBM Rational Rose, Borland Together Designer/Developer, Visio UML, Poseidon, ArcStyler, Konesa Professional, Artisan Studio, TeleLogic Tau, and MyEclipse. In addition to the above techniques, there exists additional work on model comparison approaches not covered in this paper. Two examples include the work presented by Alanen and Porres [1] and Letkeman [5].

Alanen and Porres formalize how to calculate the union and difference of UML models. They present three metamodel independent algorithms for calculating differences between two models, merging a model with the difference of two models and calculating their union. The technique assumes having repository identifiers. The paper presents a purely conceptual approach without tool support and thus is not covered further in this paper. Letkeman presents the principles of model comparison and merging in the IBM Rational Software Architect[3]. His presentation concentrates more on the practical issues related to supporting parallel development by modelers and managing model merges. IBM RSA does provide model comparison capabilities but these capabilities are proprietary for the tool.

There exists a considerable amount of related work on model differencing and comparison in general. Most of the work, however, is not applicable to UML models. For further information, the reader is kindly referred to the related research sections of the papers covered in this section.

3 Synthesis and comparison of the approaches

This section presents the key characteristics of model comparison approaches: first, background, motivation and assumptions for the approaches is discussed, followed by matching of corresponding model elements, talk on differences and their classification, extensibility, tool support and finally on their evaluation.

3.1 Background and motivation

The presented model comparison approaches share similar underlying rationale. One particularly important area benefiting from tracking changes between models is *system evolution*: to recognize the changes the system has gone through during its lifecycle. Object-oriented systems are better understood in terms of structural and behavioral models than code. There is a clear need for understanding the rationale of design evolution and software development. This line of reasoning is especially evident with Xing and Stroulia, who present their work exclusively for detecting changes between designs of subsequent versions of software written in Java.

The ability to compare models also caters for *maintaining consistency* among models produced by different developers and teams. In the case of reverse engineered models, the ability to analyze system evolution also supports maintaining consistency between design level models and implementation.

[3] http://www-306.ibm.com/software/awdtools/architect/swarchitect/

There are some differences as well. With the exception of Selonen+[4], the presented model comparison approaches have roots in version control and configuration management systems (Girshick; Kelter+; Ohst+) or software maintenance (Xing+). Consequently, they usually assume working with close-to-code design level models or even explicitly with reverse engineered Java models. Of the approaches, only Ohst+ and Selonen+ explicitly mention higher-level models. Selonen+ emphasizes comparing, merging and slicing of models, and composing and decomposing of models, and promotes creation of transient views for reasoning about the larger system models.

3.2 Assumptions on model structure

The approaches make varying assumptions regarding the nature and contents of the models they are used for comparing. In a sense, the more assumptions are made on the input models and more restrictions are based upon them, the more powerful the comparison operation can be. Xing+ explicitly states that the models to be compared are to be consecutive versions of Java software with "principled use of versioning system". Girshick claims that by making assumptions on the model structure and properties, more "semantic flavour" can be captured to the correspondence derivation rules. Indeed, Ohst+ similarly assumes that the models to be compared are revisions of a common base model stored in a model repository. Kelter+ and Girshick both discuss ensuring consistency between design models and implementation, and employ reverse engineering tools during their evaluation. Of the approaches, Selonen+ does not explicitly state assumptions on the origin and nature of the compared models (although it uses reverse engineered architecture models during evaluation as well); the authors do suggest, though, that to ensure better results on correspondence derivation, domain-specific knowledge on the nature of the model should be taken into account.

3.3 Matching corresponding model elements

Corresponding model elements are seen to represent a single conceptual entity in the compared models. Of the presented approaches, Ohst+ assumes existing repository identifiers and thus readily-available correspondence relationship; consequently, it is not covered in this section. The process of finding corresponding elements consists of *traversal* of the elements in the models to be compared and *correspondence inference* of compared model elements.

Traversal of models is most often done by moving along the whole-part or containment relationships of the model. Typically, this means first going through the packages, then classes owned by the packages, and then attributes and operations owned by the classes. Indeed, the order used by Xing+ and Girshick is divided to

[4] As the approaches lack dedicated names (or with [2] and [13], they have the *same* name), they and respective papers are hereafter referred to with the name of the first author (substituting + for *et al*) with the hope of improving readability.

three layers: packages, classes (and interfaces with Xing+), and then attributes, operations, generalizations, associations and dependencies. Kelter+ assumes a different traversal strategy than the other approaches. The models are first traversed bottom-up, after which the information on possible correspondence matches is propagated top-down[5].

In effect, the order of traversal reflects the containment hierarchy present in the UML metamodel: the composite meta-associations in the metamodel define a spanning tree for the actual UML model. This approach is followed by Selonen+ as the correspondence rules are generated directly based on the UML metamodel.

Correspondence inference, i.e. evaluation of correspondence is based on matching criteria for each pair of candidate elements. Basic criteria include the type of the element (i.e., its metaclass) and its name. Other criteria follow from the type of the element; for example, with relationships, the end elements should be corresponding as well. One way of concluding correspondence is having a degree of *similarity*: a range of certainty of the correspondence. Girshick presents a set of evaluation functions that take into account things like "location" (in element containment hierarchy), name, stereotype, type, signature (for operations) and parts (of a composite element). Kelter+ presents similarity functions per element; for each element, there is a threshold, a set of criterion and weight for each criterion. For example, similarity for classes comprises similarity of class names, ratio of similar or matched operations, ratio of similar or matched attributes, generalization target matches and package matches.

Xing+ presents overall similarity criteria comprising name similarity and structure similarity. Default name similarity metric is calculated based in terms of how many common adjacent character pairs are contained in two compared strings, while structure similarity takes into account the amount of already matched elements ("landmarks") in relation with the compared elements. An example of the latter for classes and interfaces would include the fields, methods, constructors, inner classes and interfaces they contain, together with classes and interfaces they use and are used by.

Selonen+ sees correspondence between two model elements as a binary property: if model elements correspond, they are either uniquely corresponding elements or belong to a set of candidate elements. The approach assumes that parts of the modeling language semantics are reflected by its abstract syntax. The correspondence rules based on the modeling language specification given as a MOF model (e.g. UML metamodel). By default, elements are matched based on their type and name. In addition, for each element type, their context as specified in the metamodel is taken into account. This primary context comprises the parent (owner) of an element, its mandatory neighbors (meta-associations to other metaclasses with lower multiplicity bound being greater than zero) and recursively its mandatory descendants and their mandatory neighbors; in effect, the context comprises all other modeling elements that must be present and connected with the element in question for the model to be

[5] The information given in Figure 5 of Kelter et al is unfortunately unclear. The text implies that elements are arranged according to their containment hierarchy; this, however, is contradicted by the figure showing Classifiers as leaf nodes of Attributes.

well-formed and legal. For example, in UML 1.x, associations have a parent[6] and at least two association ends (as mandatory descendants); association ends, in turn, have association as their parent, together with exactly one classifier as their mandatory neighbors. In effect, the context definition enforces the traversal order of the Selonen+ approach to follow the UML namespace hierarchy.

Table 1. Examples of identifier-independent correspondence generation.

Technique	Correspondence generation
Girshick	hierarchical matching with evaluation functions for supported model element types (name, "location", stereotype, type, signature)
Kelter+	similarity based matching (rules given for classes: classes, operations, attributes, generalizations, packages)
Selonen+	automatic generation of correspondence inference rules based on abstract syntax of the modeling language metamodel (name-type-context correspondence)
Xing+	similarity measure of element names and structure

The correspondence generation approaches are summarized in Table 1, excluding Ohst+ as it relies on the use of repository identifiers.

3.4 Classification of differences

Most of the tools aim towards change detection: automatically detecting differences between the compared models. If models are assumed to undergo evolution and changes, the differences can be classified beyond simple commonalities and differences. Both Girshick and Xing+ present the same classification of typical differences: addition of elements, deletion of elements, renaming of elements, moving of elements, cloning of elements (only Girshick), and modifying element properties. Kelter+ recognizes the same set of differences (excluding clones) with different terminology: structural differences (addition, deletion), attribute differences (renaming, modifying element properties) and move differences. Ohst+ provides a similar classification, with a more layout related focus. Selonen+ recognizes addition and deletion of elements, as well as modifying element properties[7].

[6] Type of association parent is *not* dictated by the UML; it can be *any* UML namespace. This is an example of the ambiguities that hinder tool interoperability; consequently, model comparison approaches assume that e.g. associations go under, say, certain classes. The Selonen and Kettunen approach was partly motivated by the need to ignore such ambiguities.

[7] The set operations have been used for monitoring software evolution, including moving (migration) of elements ([11], Sec. 5.5)

3.5 Extensibility and modeling language support

As pointed out by Kelter+, the correspondence inference rules and the internal representation of the models can either be document-type (i.e., diagram type) specific, or they can be configurable and extensible for different types of diagrams. Of the former, the rules presented by Girshick and Xing+ have their rational based on the types and characteristics of the modeling elements they support. Of the latter, Kelter+ reads its similarity rules from a configuration file, making the correspondence derivation in extensible. Selonen+ generates the set of rules once for a selected modeling language based on its metamodel and then uses this set of rules; the rules can be manually modified to take whatever domain specific information is available.

Apart from Selonen+ and Kelter+, the approaches assume only UML class diagrams. Kelter+ claims support for statechart and class diagrams, but only covers the latter. Selonen+ claims to support the whole of UML; the paper presents examples of class and activity diagrams, and claims to support statechart diagrams[8] as well.

Most of the techniques provide their own proprietary data representation that can be mapped to the UML metamodel. For example, Xing+ supports packages, classes, interfaces, fields, methods and dependencies, while Kelter+ supports classes, operations, attributes, generalizations and packages. Girshick supports packages, classes, generalizations, attributes, associations, and operations. Selonen+ gives examples of context information for packages, classes, interfaces, associations (and association ends), generalizations, dependencies, attributes, operations and their parameters. In addition, Selonen+ presents context information for activity diagrams with activity graphs, composite states, action states, pseudostates, final states, and transitions.

3.6 Provided tool support

To be useful, a methodology must be accompanied by a tool, supporting some input and output format, and providing a way to present the results to the user. Of the former, XMI is the OMG standard for model exchange. While there are well-documented problems regarding the use of XMI as an exchange language (e.g. [4]; also reported by Kelter+), it is the best candidate for a standard input and output format for UML models.

Of the approaches, only Kelter+ claims direct XMI support both as an input and output format. The tool is implemented as a Fujaba plugin. It includes a visualizer component that shows commonalities and differences of the compared diagrams using colors. The differencing information is also recorded in the XMI output.

The Selonen+ tool is implemented on top of a proprietary CASE tool independent UML model processing platform. The platform is integrated with IBM Rational Rose and thus enables the use of the Rose XMI importer and exporter functionality, together with Rose .mdl files. The output is a Rose model and a diagram (or a set of Rose diagrams) showing the commonalities and differences of the models using color

[8] This follows from UML 1.x statechart diagrams sharing essentially the same metamodel as activity diagrams.

coding. Alternatively, especially in the case of large models, the output can be a comma separated values text file describing the models and their commonalities and differences.

Ohst+ is implemented on top of a proprietary repository system H-PCTE with its own visualizer. Again, the commonalities and differences are shown in a unified class diagram using color coding. Of the presented approaches, the work presented in Ohst+ pays the most attention on layout and presentation issues.

Girshick uses a proprietary XML format for input models, and produces its output in SVG and HTML formats. Both output formats use color coding for showing the commonalities and differences of the models.

Finally, Xing+ tool is implemented as a part of JDEvAn Eclipse plug-in, a Java analysis environment developed by the authors. Consequently, Xing+ relies on JDEvAn to reverse engineer given Java programs as its input. Unlike the other approaches that provide graphical output in the form of a UML diagram, Xing+ generates an XML change tree shown in a dedicated Eclipse view.

3.7 Evaluation provided for the approaches

The evaluation provided for the selected model comparison approaches is summarized in Table 2. The evaluation ranges from simple class diagram comparison to analysis of real software evolution, and from as-is proof-of-concept cases to elaborate correctness and performance analysis.

The provided evaluations seem rather incommensurable and in cases of an ad hoc flavor. However, they do share some interesting commonalities. It is clearly evident that most of the approaches—actually, all of them with the exception of Selonen+—seem to have their roots in version and configuration management, and in software evolution management. The diagrams used in the evaluation have almost invariably been static structure diagrams (i.e. UML class diagrams), with the only notable exception provided by Selonen+ on UML activity diagrams.

The presented examples work with diagrams that are fall into two categories: either they are individual example diagrams, or they have been reverse engineered from Java programs. While admittedly large, the latter tend to be rather simple in structure because of the limited capabilities of reverse engineering tools; for example, even basic structures like association end multiplicities and composition relationships are rarely present in such models (see e.g. Kollman et al [4]). In addition, the expected changes in reverse engineered models are rather limited compared to design level models produced by hand.

Of the presented evaluations, only Kelter+ and Xing+ summarized the results of their experiments and discussed their quality; the former presented false negatives and positives, while the latter provided a comprehensive analysis of the results with discussion on the ‘correctness’, ‘robustness’ and thresholds of their approach together with notes on the false positives. None of the evaluations discussed the generalizability of the results, or provided analysis of the threats to internal and external validity of their respective experiments. None of the experiments included test subjects or higher level software engineering goals.

Table 2. Summary of model comparison techniques and provided evaluation.

Technique	Provided evaluation
Girshick	An example of visualizing the differences between two reverse engineered Java class diagrams. Out of 10 packages with 50 classes in total, 3 packages were selected separately for comparison. One package remained relative unchanged, while one package had gone under heavy refactoring, resulting in "many erroneously detected move operations". The result of the comparison related to the third package was presented in the paper. Presented example class diagrams contained classes, attributes, operations, associations, and generalizations.
Kelter+	Five example cases with 3+4+4+3+4 class diagrams produced from reverse engineered Java code. Number of classes per diagram varies from 17 to 267. The results were presented with the number of detected differences together with false positives and negatives. Four of the cases were analyzed quality-wise by manually inspecting the results. The performance of the tool was evaluated with regard to the given example models. Presented example class diagrams contained classes, operations, associations, and generalizations. The fifth example case with four class diagrams was reported to be a snapshot of a total run comparing 50 consecutive releases of a Fujaba package (de.uni_paderborn.fujaba.basic).
Ohst+	N/A.
Selonen+	Three example cases with correspondence rules generated based for UML 1.4.2 class diagram and activity diagram metamodels. Comparing two example class diagrams with 14 classes each, two example activity diagrams with around 10 activities each, and two reverse engineered architecture models of a Nokia mobile terminal product platform (originally described by Riva et al [9]) with 1000 components per model. Presented class diagrams contained classes, interfaces, attributes, operations, associations, dependencies, and generalizations; presented activity diagrams contained activities, states, pseudostates and transitions. Correspondence rules were also generated for UML 2.1 class and activity diagram metamodels,
Xing+	An example case comprising pair-wise comparison of 31 reverse engineered consecutive releases of a Java library (JFreeChart) reaching up to 800 classes. Analysis on time cost and similarity metrics thresholds. Detailed manual analysis on correctness and robustness of difference detection with around 6000 actual changes. The authors report additional analysis with several small-to-medium scale systems, and usage of the comparison results in further evolutionary analysis.

Further, none of the papers provided working hypotheses that could be systematically and formally evaluated. Even when restricting ourselves to monitoring the evolution of reverse engineered Java programs, and presenting the structural differences between consecutive releases, it would nevertheless be useful to go beyond mechanical comparison of diagrams and/or models and address the needs of actual software developers. The model comparison techniques should be evaluated regarding how well the tools actually support the designer in performing selected model

comparison tasks, like comprehension of the compared models, also investigating further what these tasks would actually be.

4 General qualities for model comparison approaches

Based on the previous section, some of the desirable qualities for model comparison techniques are presented. Some of the qualities are related to the methodology, while others are directly related to the implementing tools. The list is non-exhaustive but provides a starting point for evaluating model comparison techniques and tools.

Identifier independence denotes the ability to infer corresponding model elements without the presence of repository identifiers. To reason about models, a correspondence relationship needs to be established between model elements representing the same concepts. While such a correspondence relationship is usually intuitively clear, it is often implicit in the models. In such a case, there is a need for bringing the correspondence relationship between UML model elements explicit. It is argued that if a model comparison technique relies solely on a common repository with unique identifiers, it restricts the applicability of the tool. Means of inferring correspondence were presented in Section 3.3; they included using identifiers (element names), structure (composite elements, neighbors), and element similarity (string similarity, synonyms, structure similarity, etc.).

Customizability denotes the possibility to customize the approach to a particular domain. The user should be able to tune the correspondence inference and similarity functions for a particular domain with whatever additional information is available. Examples of customization include augmenting the correspondence relationship generation with additional, domain-specific rules, affecting the reconciliation strategies and merging preferences, and tuning the visualization strategies. Means of supporting customizability include allowing the user to affect the correspondence generation rules by e.g. providing access to the rule configuration files or by allowing the user to add rules to the abstract syntax based on which the rules are generated. This quality follows from issues presented in Section 3.2, 3.3, and 3.5.

Reliability denotes the ability of a model comparison approach to adhere to a certain level of reliability. Reliability can be evaluated in regard to e.g. the correspondence inference, detection of commonalities and differences, model reconciliation, and model merging. The approach should produce reliable results within certain limits, and also enable the user to spot the potential false negatives and false positives, and understand issues like merge rationale. Reliability was discussed in Section 3.7.

Extensibility denotes the ability of an approach to support a range of diagram types and model elements. Approaches should be extensible to support larger modeling language subset. Most approaches require the designer to extend the tool in order to implement support for additional modeling elements. Solution to extensibility include providing the user with access to a configuration file containing the rules (Kelter+)

and generating the rules directly based on the abstract syntax of the modeling language subset (Selonen+); both approaches in principle relieves the designer from touching the tool itself. This quality is complementary to customizability. Extensibility was discussed in Section 3.3 and 3.5.

Usability issues, e.g. means of supporting user interaction, usability and acceptance can include providing dialogs and/or other means for selecting the source and target models and for setting the parameters (related to customizability). While qualities like reliability are important, effort must also be used to make the tool usable and acceptable for the user. If human-computer interaction issues like usability, acceptability, likeability, effectiveness, learnability, and flexibility are ignored, the tool is not likely to be used. Usability issues to consider include visualization of the commonalities and differences of the compared models, conflict and inconsistency reporting, navigability (e.g. navigating between generated model and input models), and ability to affect e.g. model reconciliation during merging. Usability would be important part of tool support discussed in Section 3.6 and evaluation in Section 3.7; however, none of the included approaches (with the possible partial exception of Ohst+ emphasizing presentation issues) paid attention to usability.

Composability is a system design principle that deals with the inter-relationships of components. It denotes the ability to use the tool as a part of a larger model processing framework. In principle, a highly composable system would provide recombinant components that can be selected and assembled in various combinations to satisfy specific user requirements. Composability could also be seen to cover the input and output formats supported by the tools, i.e., their ability to be integrated with existing model processing platforms. Of the included approaches, Xing+ is integrated with Eclipse, Selonen+ with IBM Rational Rose, and Kelter+ with Fujaba. Composability is covered in Section 3.6.

Non-intrusiveness can be defined as the technique's ability to work and manage models in a non-intrusive way. The tool should be able to produce views to the system that do not affect the existing models. It is important that the operations do not touch the model repository unless that is explicitly required by the user. This is particularly important when working with production models. The output model is kept separate from the input models and written e.g. under a new package, with all relevant model elements copied into the namespace. Again, being a tool related quality, issues related to non-intrusiveness is covered in Section 3.6.

This section listed seven quality attributes that stem from the issues covered in Section 3. Of the attributes, identifier-independence and reliability can be seen as qualities related purely to methodology; customizability and extensibility are mostly related to methodology but have a tool component as well; and finally, usability, composability and non-intrusiveness are purely tool related qualities. The list of qualities is by no means comprehensive or complete, but it offers a starting point for discussing about the qualities a model comparison tool should adhere to. There can naturally be other qualities as well, like attainability: the tool should be available for

installing and downloading, i.e., it should be freely downloadable, easy to install, and usable on a widely-used tool.

5 Usage scenarios for model comparison approaches

This section lists some potential usage scenarios for model comparison techniques. Some of them have already been research goals for the techniques summarized in Section 2. The rationale for the presented usage scenarios stem from the observation that correspondence derivation is a key part of the approaches covered in this paper. It can be argued that model differencing is only one of a variety of usage scenarios for such *correspondence based approaches* for reasoning about models. As with quality attributes, the presented list of usage scenarios is not exhaustive, but aims to provide a starting point for discussion.

Model merging is used for integrating the modeling artifacts produced by several designers or design teams. Similarly, the increments produced during a software development process can be integrated into the existing system model. As an example of such a process, consider a use case driven, incremental software development process producing an initial structure model (a class diagram) and for every use case, a set of sequence diagrams. The structure implied by the sequence diagrams can be transformed into a class diagram and merged together with the existing system model through model merging. Model integration is explicitly mentioned as a research goal by Kelter+ and Selonen+. In order to support cooperative team work a version management system which supports UML models is absolutely necessary, as is the ability to calculate differences, present them to developer and provide merge operations to come to a consistent model.

Model comprehension can be supported by allowing the designer to create transient views for exploring the parts of the system she is interested in. Designers and stakeholders use different diagram types for exploring the system from different viewpoints or levels of abstraction. The ability to compose together larger units of merge, slice and compare functionality can be used for generating new views that contribute to a given viewpoint or stakeholder interest. To this end, model composition and decomposition supports the construction of models from model fragments, each describing individual concern or a cross-cutting aspect of the system. The roots of Selonen+ approach lie in the need for ability to generate transient views, slice models against each other, integrate them, and highlight their commonalities and differences provide a powerful tool for generating documentation.

Inconsistency detection can be supported by model differencing. Inconsistencies can occur among corresponding model elements (e.g. abstract vs. concrete class, inconsistent association end multiplicities); depending on the source of the compared models, differences can pinpoint other types of inconsistencies as well (e.g. missing association, conflicting inheritance hierarchies). Inconsistency detection can also be applied in the context of reverse engineering: by comparing reverse engineered

models against design models the designer can detect potential implementation level mismatches. The latter is addressed by a majority of the included approaches.

Software evolution analysis, i.e. comparing subsequent versions of design models is an obvious usage scenario for model comparison techniques. One specific evolution topic is model refinement and refactoring; with model refinement, matching of corresponding modeling elements is not straightforward. For example, model elements may be refined into other model elements. If the refinements are done step-wise, model differencing provides a potential way of tracing refinement relationships. Of the covered approaches, both Selonen+ and Kelter+ discuss monitoring evolving software systems; however, Xing+ goes into depth in using the gathered difference information to elicit more complicated structural change patterns like refactorings, and for collecting project-specific evolution knowledge.

In addition, Model Driven Development is another paradigm that can benefit from model integration**.** With support for composability and model integration, the model comparison tools can be seen as an important operation for supporting Model Driven Development for merging together the results of different transformations. Merge is a fundamental operation for building a modeling framework supporting chains of model manipulation operations.

6 Concluding remarks

This paper gave a review of five available UML based model comparison approaches. In a nutshell, a model comparison approach can be defined as comprising the following steps: 1) inference of corresponding model elements, 2) detection of commonalities and differences between models, and 3) presenting the commonalities and differences to the user. The paper presented a synthesis of the approaches, compared their commonalities and differences, presented a list of qualities they should adhere to, and finally suggested some potential usage scenarios for them.

It can be argued that model differencing is only one of a variety of usage scenarios for such correspondence based approaches for reasoning about models, and therefore this work should be taken further to support a larger software engineering context. Further, the model comparison techniques should be evaluated regarding how well the tools actually support the designer in performing selected model comparison tasks, like comprehension of the compared models, also investigating further what these tasks would actually be.

Acknowledgments. The author wishes to thank professor Tarja Systä for her valuable comments.

References

[1] Alanen, A., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.): Proc. UML 2003 – The Unified Modeling Language. Lecture Notes in Computer Science, Vol. 2863. Springer-Verlag, Berlin (2003) 2–17

[2] Girschick, M.: Difference Detection and Visualization in UML Class Diagrams, Technical University of Darmstadt Technical Report TUD-CS-2006-5, August (2006)

[3] Kelter, U., Wehren, J., Niere, J.: A Generic Difference Algorithm for UML Models. Software Engineering (2005) 105–116.

[4] Kollman, R., Selonen, P., Stroulia, E., Systä, T., Zundorf, A.: A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In: Proc. WCRE 2002. IEEE Computer Society, Washington DC (2002)

[5] Letkeman, K.: Comparing and merging UML models in IBM Rational Software Architect, IBM Rational, July (2005) On-line at http://www-128.ibm.com/developerworks/rational/library/05/802_comp3/. Last accessed 13th July 2007.

[6] Object Management Group. Unified Modeling Language: Superstructure, version 2.1, April 2006. On-line at http://www.omg.org/cgi-bin/doc?ptc/2006-04-02.

[7] Object Management Group, Meta Object Facility (MOF) Specification Version 1.4, 2002. On-line at http://www.omg.org/cgi-bin/doc?formal/02-04-03.

[8] Ohst, D., Welle, M., Kelter, U.: Differences Between Versions of UML Diagrams, In: Proc. of ESEC'03, Helsinki, Finland (2003) 227–236

[9] Riva, C., Selonen, P., Systä, T. Xu, J.: UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance. In: Proc. of ICSM'04, Chicago, IL, USA. IEEE Computer Society, Washington DC (2004) 50–59

[10] Selonen, P.: Set Operations for the Unified Modeling Language. In: Kilpeläinen, P., Päivinen, N. (Eds.): Proc. of SPLST 2003, Kuopio, Finland. University of Kuopio (2003) 70–81

[11] Selonen, P.: Model Processing Operations for the Unified Modeling Language. PhD Thesis, Tampere University of Technology. (2005)

[12] Selonen, P., Kettunen M.: Metamodel-Based Inference of Inter-Model Correspondence. In: Proc. of CSMR'07, Amsterdam, Holland. IEEE Computer Society, Washington DC (2007) 71–80

[13] Xing, Z. and Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: Proc. of ASE'05, Long Beach, CA, USA. ACM Press, New York NY (2005) 54–65

Dual Data Model for Metadata: Combination of Relational Model and RDF Model

Timo A. Kosonen, Ilkka Salminen, and Tommi Lahti

Nokia Research Center, P.O. Box 1000,
33721 Tampere, Finland
timo.kosonen@tut.fi, {ilkka.salminen, tommi.lahti}@nokia.com

Abstract. This paper presents a dual data model for metadata in a networked, distributed, and ubiquitous environment from where we collect content, context, relationship, usage history, and community metadata. We needed a data model that fits with our requirements for metadata collection and created a two layered or two-tier data model combining relational and Resource Description Framework (RDF) data models. We tried the created dual data model in a system collecting music metadata. We learned that our metadata model fits well with our requirements. The described system can be implemented with the current technology and it supports advanced queries based on meta-metadata.

Keywords: Data models, metadata, databases.

1 Introduction

1.1 Research Context

Our vision is to manage a huge amount of electronic media content in a networked, distributed, and ubiquitous environment of mobile devices over a long period of time (Fig. 1 shows the basic environment). Metadata is an essential part of the process [14]. It is not possible to manage media content effectively without content-related metadata [15]. We think that context, relationship, usage history, and community metadata are as important. We collect metadata from all over the network storing it into a huge evolving database. This database is updated and queried constantly in real time.

There are five types of metadata we are interested in: content, context, relationship, usage history, and community metadata (see Table 1 for examples of different types of metadata). Content metadata relates to the media content itself. It includes the features that originate from the content data, via signal analysis methods, for example. Context metadata describes the context where the media content is created, stored, or consumed. It includes the location of watching or listening to a media track, for example. Relationship metadata describes relationships between different media content items. Usage history metadata describes how and when the content has been

consumed. Community metadata describes communities that create, store, or consume media content and it includes different groups of people such as friends, family, colleagues, contacts, and so on.

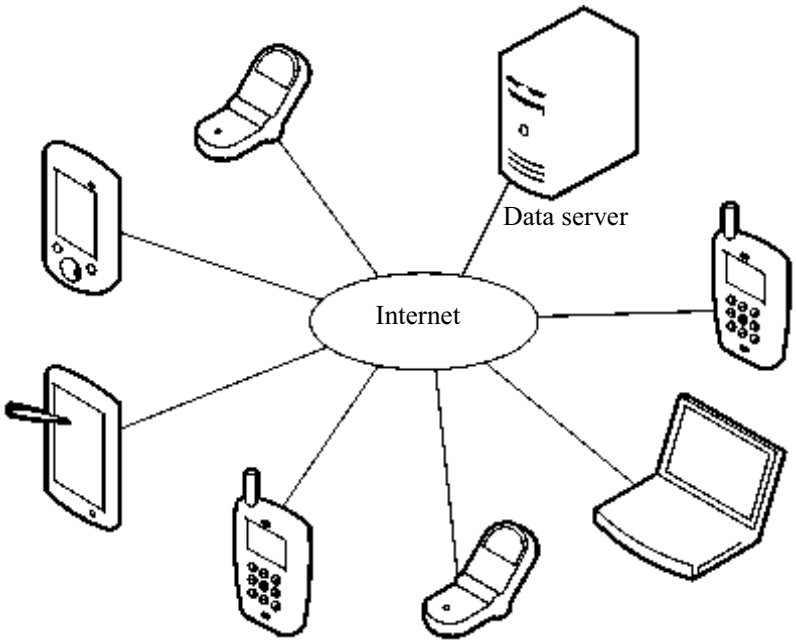


Fig. 1. The environment of our research containing various mobile devices connected to a data server via Internet.

1.2 Problem Description

Collected metadata is not always useful as such, so we need to analyze the metadata and generate meta-metadata (see [10] for a discussion about metadata on different hierarchical levels) from the original metadata. This analysis could include data clustering [9], for example.

In order to collect metadata, there must be an existing data model instance for the representation and use of the metadata. On the other hand, only after collecting metadata and studying the metadata, we can really understand what kind of data model is suitable for it.

The problem is how to build a data model for metadata that is flexible enough to enable to:

- collect a huge amount of metadata
- store lots of different types of metadata, even beforehand unknown types
- collect metadata without finalizing the data model for metadata analysis results

- use previously unknown data analysis algorithms in the future
- query the analysis results through the same interface

1.3 Previous Approaches

There are several examples where media related metadata has been stored on top of a relational database. For example, Tsinaraki et al. developed their metadata framework on top of relational databases [17], and Cano et al. presented a common repository including metadata on top of relational databases [2]. The architecture of Tsinaraki et al. also includes interfaces for querying metadata stored in their system [17].

1.4 Our Approach

We propose a dual data model for metadata containing two distinct layers or tiers. The first layer based on the relational data model is meant for an initial collection of metadata into a relational database. This layer serves the purpose of collecting as much different types of metadata as possible. At this phase we are not interested in the relationships between different metadata items, but in collecting as many of them as possible.

The second layer based on the Resource Description Framework (RDF) data model [13] stores the results of the analysis of the metadata collected on the first layer. The second layer demands a more flexible and general structure than the first layer and is suitable for multiple different data analysis algorithms enabling changes and modifications. At this phase, we focus more on the relationships between different metadata items than the metadata items themselves.

Our main contribution is the dual data model described above enabling simple continuous collection of metadata on the first layer and providing a flexible abstraction for data analysis results on the second layer. In addition, we have established our own categories for metadata.

1.5 Paper Content

In this section, we introduced the context of this paper, defined the problem, and introduced our approach to the problem. The second section describes the background of the research area of the paper. The third section explains our approach in more detail. The fourth section discusses our implementation. The fifth section contains a case study showing how our approach suits well to collecting music related metadata. The sixth section compares our work to related work, and the seventh section presents the conclusions.

2 Background

2.1 Data Collection and Creation

Data collection and creation is a profitable business. The collected data is used for advertising and other knowledge discovery and data mining purposes. Internet search engines such as Google collect data with the help of web spiders and from people's queries. The Internet search engines use large scale databases of indexed Internet content and provide people with free search capabilities helping the Internet to become an integrated part of people's lives.

Mobile phones have a fundamental difference in data collection when compared to conventional computers such as desktop computers and laptops. People carry mobile phones most of the time keeping the power on, but desktop computers and laptops are generally left home for leisure time or used at work. Mobile phones with an Internet connection provide a greater potential for collecting metadata than conventional computers. Above all, mobile phones are *context aware* inherently. For example, they need to track their carrier's position in order to function.

2.2 Metadata and Meta-metadata

Metadata is data about data. Essentially, metadata describes something that can be called data. Metadata about data is on a higher abstraction level than the data itself. For example, a book catalog in a library contains metadata about the books of the library. See [6] for an introduction to metadata.

There are different types of metadata and views on metadata. We think of metadata in relation to media content, so we have established our own categories for metadata. See Table 1 for examples of different types of metadata organized according to our metadata categories.

Content Metadata, Content metadata describes media content. Content metadata can be deduced from the content itself requiring no information from outside the content. For example, the content type of .mp3 files is audio/mpeg. Usually, an ID3 container in conjunction with an mp3 file includes information about the title, artist, and album of the file, if they cannot be deduced from the filename. Different signal analysis methods can be used for analyzing tempo, loudness, and other musical features from musical content. The analyzed metadata can be used, for example, for media recommendation or advanced media navigation such as in [11].

Context Metadata, Context metadata is information that describes the situation of an entity, for example, the media content or a person [4]. The location and movement of the user who consumes content is context information that describes how the content is used. This information cannot be derived directly from the content itself but is valuable metadata about the content. It could also be interesting to know other context information, for example, whether a user wears a headset when consuming the content or whether the user consumes the content at night.

Relationship Metadata, In addition to context metadata, there is other metadata that cannot be deduced from the content itself. Relationship metadata describes how a single media content item is related to other content items. For example, relationship metadata can be used to describe that a music track relates to other music tracks in the same playlist, and a certain image is album art for the music track. These relationships can be defined across media types.

Usage History Metadata, Third important type of metadata is usage history metadata describing how and when the content is used. In its simplest form, it consists of play counts keeping track on how often media content is played. Generally, usage history metadata just describes how the media content is used. Usage history provides lots of information about how valuable the content is to its holder.

Community Metadata, Metadata can also describe more than just attributes of any given media content. It can describe relationships between different users consuming media content. This metadata may be called community metadata, because it relates to user communities. Friends may have a similar taste for media content, whereas colleagues may differ in their taste for media content. Fig. 2 shows the relationships between content items, context, users, and different types of metadata.

Table 1. Examples of different types of metadata for each of our metadata category.

Content metadata	Context metadata	Relationship metadata
Content type	Location	Album order
Content creator	Course	Playlist order
Title	Cell ID	Album cover art
Artist	Volume	Soundtrack vs. videotrack
Album	Headset state	Different versions
Tempo or speed	Repeat mode	Required certificates
Loudness	Shuffle mode	Similarity
Musical key	Lightness	
Musical meter	Noise level	
Musical genre	Temperature	Other related content

Usage history metadata	Community metadata
Time and play counts	Friends
Share counts	Family
Downloaded from	Colleagues
Uploaded to	Contacts
Emailed to	Acquaintances
Received from	People nearby
Mixed with	People in the same town
Edited	People in the same country
Updated	Consumers of the same media
Deleted	Consumers of the same genre

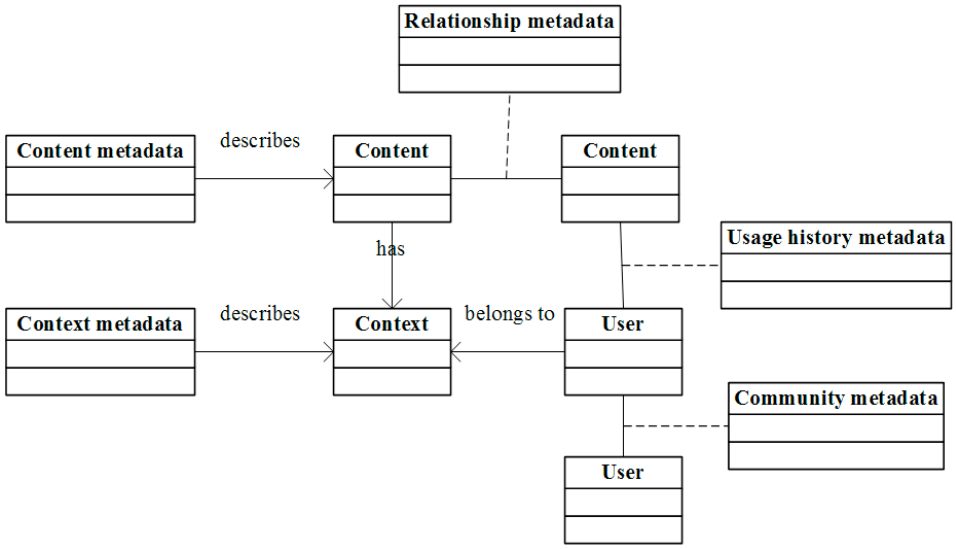


Fig. 2. The relationships between content items, context, users, and different types of metadata.

Meta-metadata, Meta-metadata is data about data about data. Meta-metadata could be generated by running analysis algorithms on metadata, for example. In our system, we could store the queries made of metadata on our database. The stored information about the queries is meta-metadata for the original media content. Because the same methods can be used for utilizing meta-metadata as metadata, the distinction between meta-metadata and metadata is rather informative highlighting complexity or the existence of another dimension. See [10] for a discussion about metadata on different hierarchical levels.

2.3 Relational Model

Relational model was first introduced in [3] to protect database users from having to know the internal representation of data in databases. Since then, relational databases have become popular for storing almost any kind of data. In a relational model, each distinct relation (R) is stored as a row into a table. The ordering of the rows does not matter. The columns or sets (S_n) of the table are ordered and each of them has a label. Table 2 shows an example table according to a relational data model.

Table 2. Example table called "track" that conforms to the relational data model.

	id (S_1)	title (S_2)	artist_id (S_3)	time_added (S_4)
R_{345}	345	Favourite Worst Nightmare	26	21/05/2007 15:54
R_{173}	173	Stadium Arcadium	16	21/05/2007 15:54
R_{842}	842	Because Of You	94	21/05/2007 15:54
R_{963}	963	Volta	56	21/05/2007 15:54
R_{972}	972	Call Me Irresponsible	76	21/05/2007 15:54

The relational data model includes primary keys and foreign keys. A primary key refers to the combination of sets that define uniquely each relation. A foreign key provides a cross-reference to another relation. Database normalization is another important concept of the relational data model where various normal forms are sought.

Structured Query Language (SQL) has become a standard in ANSI and ISO for manipulating and retrieving data stored in relational databases.

Example SQL query

```
SELECT title FROM track
WHERE id < 1000
ORDER BY title
```

2.4 RDF Model

The World Wide Web Consortium (W3C) published a recommendation in 1999 containing the Resource Description Framework (RDF) model [13]. RDF has now become a specification family providing "a lightweight ontology system to support the exchange of knowledge on the Web" [8].

RDF model contains four different sets: Resources, Literals, Properties (subset of Resources), and Statements. Each statement is a triple of the form: {Predicate, Subject, Object}. Predicate belongs to Properties, Subject belongs to Resources, and Object belongs to Resources or Literals. Fig. 3 shows an illustration of an example statement.

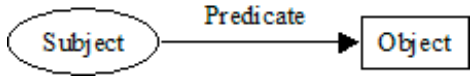


Fig. 3. Resource Description Framework (RDF) statement consists of Predicate, Subject, and Object.

The Fig. 3 can be interpreted so that Object is the value of Predicate for Subject. An alternative interpretation is that Subject has a property Predicate with a value Object. Alternatively, Predicate of Subject is Object.

Thus, RDF data consists of triples. There exist several systems that store RDF triples in a relational database and even a system that stores RFD data as a graph in an object-oriented database [1].

3 Two Layer Data Model

Our proposal is to store metadata using a dual data model containing two distinct layers or tiers. This double layer structure allows us to collect a huge amount of different types of metadata without finalizing the data model for metadata analysis results. Fig. 4 shows our system.

3.1 First Layer

First, metadata is collected over the Internet from mobile devices. This collection happens automatically by special software or semi automatically by asking questions from the user. The user's device is connected to a data collection server through a dedicated interface. The server has a relational database containing a relational data model for the collected data. At this point, we do not need to know about what we are going to do with the collected metadata in the future. We are only interested in collecting as much metadata as possible.

Relational data model is a well known model and well suitable for collecting this type of data. We can easily design the database using techniques associated with the relational data model such as using primary keys, foreign keys, normalization, and indices. This design process is easy, because we do not need to care about relationships between collected metadata or their semantics. All collected metadata is just stored into the relational database.

3.2 Second Layer

When the amount of data reaches a certain point, we can start to analyze the metadata with different data analysis algorithms, for example data clustering. We expect to use many different kinds of algorithms, so we need a flexible data model for storing the analysis results. We need to be able to query the analysis results regardless of what analysis algorithms we use or what results we get. We chose the RDF data model for storing the analysis results and enabling this kind of querying. RDF data model also enables creating new relationships between Subjects easily and dynamically, even if we do not know the type or nature of those relationships beforehand.

The second layer also stores possible meta-metadata. For example, third parties may query the data stored in the RDF data model and the data about these queries may be stored in the RDF data model too. This data could consist of weight coefficients and data that describe the importance of the other data. This information can then be further queried.

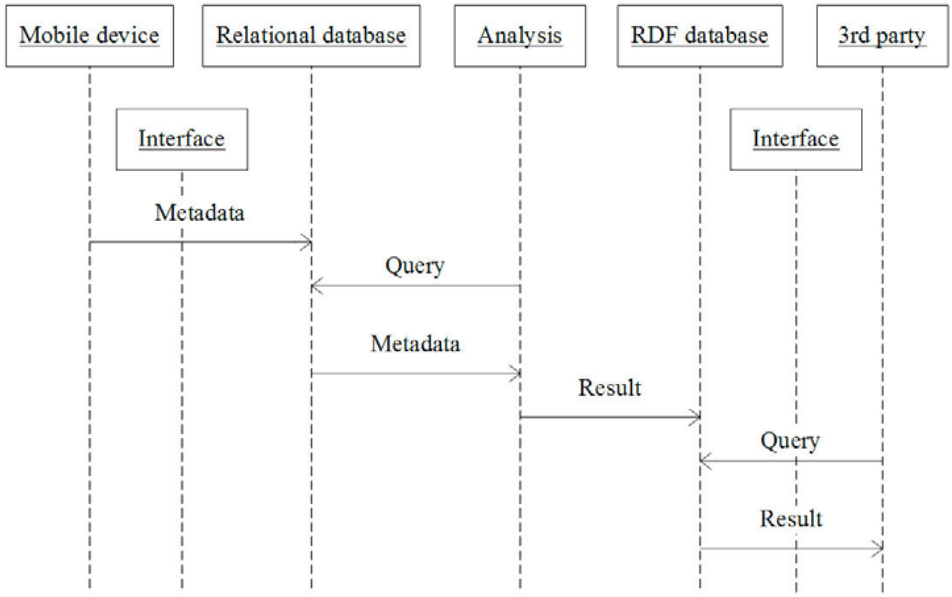


Fig. 4. Sequence diagram showing our dual data model and the overall system.

RDF enables forming triple graphs. Triple graphs are flexible graphs that can be transformed by triple graph grammars [12]. This ability increases the flexibility of the RDF data model and allows us to possibly modify analysis results automatically into a more understandable or queryable format.

Because all analysis results will be stored into a coherent triple format, we are able to design simple APIs for third party applications interested in querying the data. In such a case, the same API can be used for querying different types of metadata by different types of applications.

4 Implementation Details

4.1 SOAP Container

We chose to use SOAP [7] for transferring collected metadata from mobile devices to our server over the Internet. We send SOAP messages with HTTPS queries so firewalls should not pose problems to our system and the traffic should be secured reasonable well. The following unicode XML code shows the syntax for the SOAP messages that we use.

The syntax of the SOAP messages that we use containing two metadata structs

```
<?xml version='1.0' encoding='UTF-8'?>
<env:Envelope xmlns:env="http://. . .">
  <env:Body>
    <struct_name_1 id='123' request_id='456'
      time='2007-03-15T14:38:11+02:00'
      user_action='1'>
      <first_struct_item_1>data</first_struct_item_1>
      <second_struct_item_1>data</second_struct_item_1>
      <third_struct_item_1>data</third_struct_item_1>
    </struct_name_1>
    <struct_name_2 id='124' request_id='457'
      time='2007-03-15T14:38:11+02:00'
      user_action='0'>
      <first_struct_item_2>data</first_struct_item_2>
      <second_struct_item_2>data</second_struct_item_2>
      <third_struct_item_2>data</third_struct_item_2>
    </struct_name_2>
    . . .
  </env:Body>
</env:Envelope>
```

4.1 Metadata Structs

Each SOAP message may contain multiple metadata structs. Each struct has an identifier (id) and optionally a request identifier (request_id) that are used for identifying structs and mapping replies to the right requests. Each metadata struct has two attributes: a time stamp and a boolean describing whether the metadata is initiated by the user. The structs themselves may contain an arbitrary number of variable members.

We specified several different metadata structs. There is an example of our location metadata struct below. The syntax is similar to C or C++.

Example metadata struct that contains some location metadata related to the location where the user consumes media content excluding Cell ID and relative location information such as "home", for example.

```
/* Location
Whenever the location of the device changes over a
threshold distance (set by Treshold, or is the default
threshold if not set), the application sends this
struct to the server. The right response from the
server is E_OK.
*/
struct Location {
  double latitude;
  double longitude;
  double altitude; // meters
  double horizontal_accuracy; // meters
  double vertical_accuracy; // meters
};
```

4.2 Server

Our server is a Linux server containing a relational database. We use MySQL, and SQLObjects. Our server is capable of storing the RDF data model into the relational database.

5 Case Study: Music Metadata

This section describes a case study where we applied our approach to music metadata.

5.1 Application Overview

We evaluated our approach by designing how it will be used for collecting, analyzing, and distributing music metadata. Our plan is to collect music metadata for allowing different services such as music recommendation services to query the collected metadata. Fig. 5 shows the database schema of our relational database. This database is meant for collecting music metadata from mobile devices before any data analysis. This database has been implemented and it is already in use.

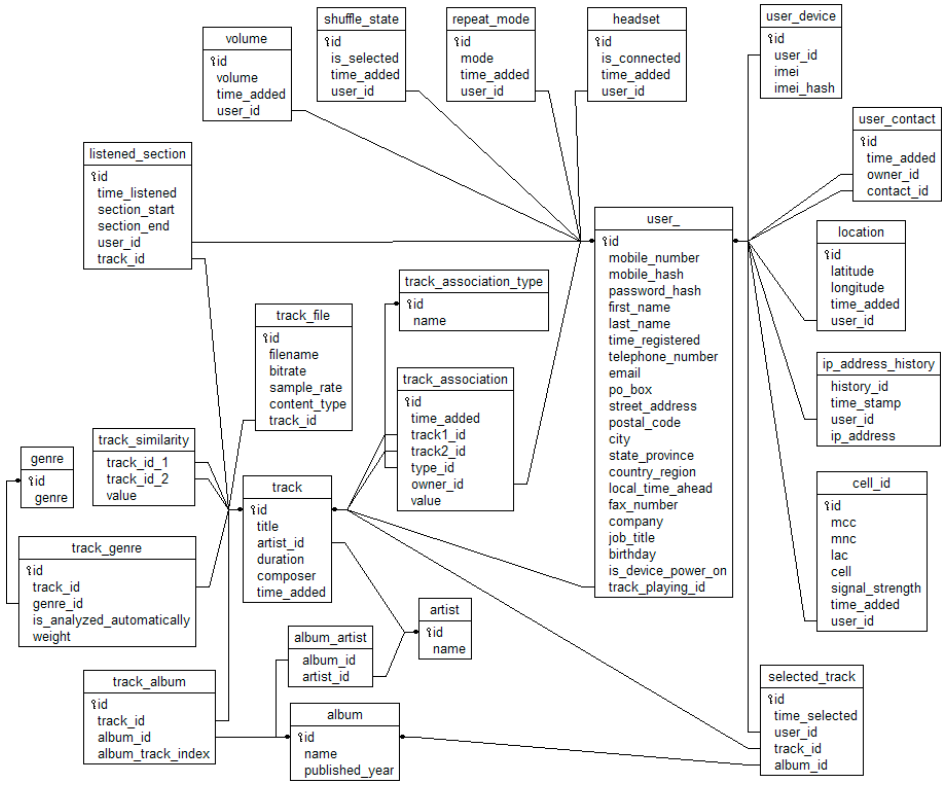


Fig. 5. Our relational database schema of collecting metadata of users' music consumption.

We will use the collected metadata as a test set for trying different kinds of data analysis algorithms such as clustering. We will store the data analysis results into RDF triples. This model allows us to possibly transfer the results with triple graph grammars into a more queryable format and use a single and simple API for querying multiple different types of metadata.

One example about data analysis results is the distance measures between different music tracks. Fig. 6 shows how a distance between two music tracks is stored into an RDF triple.

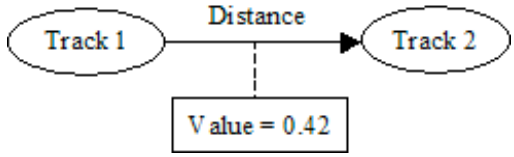


Fig. 6. Example RDF triple storing a distance measure between two music tracks.

5.2 Benefits

Designing the relational data model for collecting music metadata was straightforward using conventional means. We were able to focus on problems on the application area, because the supporting system was set up easily. The same was true for the RDF data model. We were able to store the RDF model into the same relational database with the relational data model.

The server is scalable and we will be able to move the database into another very large database if needed. The HTTPS protocol, which we use for transferring metadata from mobile devices to the server, is suitable for small and large systems alike.

6 Related Work

Others have also collected metadata relating to media content, especially relating to the context metadata at the time of capture.

In [2], the authors described a common database of audio, metadata, ontologies, and algorithms that they implemented. They do not deal with the database architecture, but state that the tools and functionalities do not depend on it. We have approached the problem from a different direction by considering what kinds of data models are most suitable for the collection and analysis of metadata.

In [17], the authors described how they also implemented their metadata framework on top of relational databases. Their data model is closely based on the MPEG-7 metadata model. We did not want to follow the same path, because there may be insufficient amount of application that would benefit from the use of MPEG-7 metadata [16]. The authors also enabled queries on the stored metadata.

In [15], the authors described how they developed a metadata creation system for images from mobile devices. They used an object-oriented database for storing the created metadata. Our goal is to allow a fast collection of lots of different types of metadata without a more complicated implementation, so we adhered to a conventional relational data model on our first layer.

In [5], the authors described their mission to create terascale collections of music related data. They did not specify the underlying data models.

7 Conclusions

We described a two layer metadata model for collecting different types of metadata in a networked, distributed, and ubiquitous environment. We combined a well known relational data model with RDF data model. We applied the combined data model to collecting music metadata. We observed that the combined metadata model fits well to our requirements. We showed that the described system can be implemented with the current technology and it supports transfers of data analysis results and advanced queries based on metadata and meta-metadata.

Our work is still on a preliminary stage allowing us to do more research on related areas in the future. In particular, we are interested in different metadata analysis algorithms for music recommendation, for example, and a more detailed implementation of the data model on the described second layer collecting the results from the metadata analysis. We strive to define our data model for metadata and meta-metadata further when more requirements are known.

Acknowledgments. This research has been funded by Nokia Research Center. We wish to thank Prof. Dr. Kai Koskimies for his comments and Mr. Marko Takanen and Mr. Jarno Seppänen for their ideas that they raised during the implementation.

References

1. Bönström, V., Hinze, A., Schweppe, H.: Storing RDF as a Graph. In: Proceedings of the First Latin American Web Congress, LA-WEB 2003, November 10-12. IEEE Computer Society, Washington, DC (2003) 27–36
2. Cano, P., Koppenberger, M., Ferradans, S., Martinez, A., Gouyon, F., Sandvold, V., Tarasov, V., Wack, N.: MTG-DB: A Repository for Music Audio Processing. In: Proceedings of the Fourth International Conference on Web Delivering of Music, WEDELMUSIC 2004, Barcelona, Spain, September 13–15. IEEE Computer Society, Washington, DC (2004) 2–9
3. Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. In: Communications of the AMC, Vol. 13, Number 6, June. ACM Press, New York, NY (1970) 377–387
4. Dey, A. K.: Understanding and Using Context. In: Personal and Ubiquitous Computing, Vol. 5, Issue 1. Springer-Verlag (2001) 4–7
5. Downie, J. S. Futrelle, J.: Terascale Music Mining. In: Proceedings of the ACM/IEEE SC 2005 Conference on Supercomputing, Seattle, WA, November 12–18. IEEE Computer Society, Washington, DC (2005) 71–72
6. Gill T., Gilliland-Swetland A., Baca M.: Introduction to Metadata, Pathways to Digital Information, Getty Research Institute (1998)
7. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J-J., Nielsen, H., Karmarkar, A., Lafon, Y.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C (2007)
8. Herman, I., Swick, R., Brickley, D.: Resource Description Framework (RDF), http://www.w3.org/RDF/, accessed in May (2007)
9. Jain, A. K., Murty, M. N., Flynn P. J.: Data Clustering: A Review. In: ACM Computing Surveys, Vol. 31, No. 3, September. ACM Press, New York, NY (1999)
10. Kerhervé, B., Gerbé, O.: Models for Metadata or Metamodels for Data? In: Proceedings of 2nd IEEE Metadata Conference, Silver Spring, Maryland, September 16–17. IEEE Computer Society (1997)

11. Kosonen, T. A., Eronen, A. J.: Rhythm Metadata Enabled Intra-Track Navigation and Content Modification in a Music Player. In: Proceedings of the 5th International Conference on Mobile and Ubiquitous Multimedia, Stanford, CA, December 4–6. ACM Press, New York, NY (2006)
12. Königs, A.: Model Transformation with Triple Graph Grammars, In: Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica (2005)
13. Lassila, O., Swick, R. R.: Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22 February. W3C (1999)
14. Lehikoinen, J., Aaltonen A., Huuskonen P., Salminen I.: Personal Content Experience, Managing Digital Life in the Mobile Age, John Wiley & Sons, Ltd, England (2007)
15. Sarvas, R., Herrarte, E., Wilhelm, A., Davis, M.: Metadata creation system for mobile images. In: Proceedings of The Second International Conference on Mobile Systems, Applications, and Services, Boston, MA, June 6–9, 2004. ACM Press, New York, NY (2004) 36–48
16. Stamou, G., van Ossenbruggen, J., Pan, J. Z., Schreiber, G., Smith, J.R.: Multimedia Annotations on the Semantic Web. In: Multimedia, Issue 1. IEEE Computer Society (2006) 86–90
17. Tsinaraki, C., Fatourou, E., Christodoulakis, S.: An Ontology-Driven Framework for the Management of Semantic Metadata Describing Audiovisual Information. In: Advanced Information Systems Engineering. Lecture Notes in Computer Science, Vol. 2681. Springer-Verlag, Berlin Heidelberg New York (2003)

Comparing two Implementations of an Approach for Managing Variability in Product Line Construction Using the GMF and GME Frameworks

Hugo Arboleda[1,2], Rubby Casallas[1], Jean-Claude Royer[2]

[1] Software Construction Group, University of Los Andes, Carrera 1 N° 18A 10, Bogotá, Colombia
[2] OBASCO Group (INRIA & LINA), Ecole des Mines de Nantes, La Chantrerie, 4 rue Alfred Kastler, 44307 Nantes, France
{hugo.arboleda, jean-claude.royer}@emn.fr, rcasalla@uniandes.edu.co

Abstract. In this paper, we present a comparison of two implementations of our proposed MDA approach for managing variability in a software product line. The implementations correspond to two representative frameworks based on the Model Driven Engineering (MDE) principles. These frameworks are the Graphical Modeling Framework (GMF) and the Generic Model Environment (GME). We built the core assets of the product line and we generated applications using the two different frameworks. The core assets that we built are: feature models, metamodels, *mapping* models, and three different types of transformation rules. We built the transformation rules using two different languages: the ATLAS Transformation Language (ATL) in the context of GMF and, the Embedded Constraint Language (ECL) in GME.

Keywords: Model Driven Architecture, Variability, Software Product Lines, and Model Transformation.

1 Introduction

A software product line (SPL) is a set of software systems that satisfies specific needs for a segment of the market. In a SPLs development approach, we can create new products of the line reusing components called core assets. While the creation of core assets is part of the domain-engineering process, the creation of applications reusing the core assets, is part of the application engineering process [1]. The management of variability in an SPL includes: (1) how to express the common and variable characteristics of a SPL, and (2) how to build applications that include common characteristics, and a subset of the possible variable characteristics. Currently, feature models are a standard *de facto* used as a mechanism to support variability.

Model Driven Architecture (MDA) [2] is an approach of generative reusability. In MDA the separation of domains and its generative nature make it a useful approach for the creation of SPLs. Results of recent researches show how the use of MDA in conjunction with SPL engineering (MD-SPL) facilitates the definition of SPL creation processes [3] [4].

The one-step model-to-model transformation process (PIM to PSM), suggested by MDA, does not take into account the separation of domain concerns, *i.e.* the ability to

identify, modeling, and manipulate those parts of software systems relevant to a particular domain (*e.g.* business logic, multi-level architecture design, programming language). Separation of domain concerns is important in MD-SPL to refine high-level design models gradually until obtaining low-level design models. We presented a scheme in [5] to manage the variability in SPL construction processes using an MD-SPL approach. In our scheme, we separated concepts related to product lines in different domains: (1) the business logic domain, (2) the architecture domain, and (3) the platform technological domain. We created metamodels, feature models, *mapping* models, and transformation rules as core assets for each domain. This scheme enables us to transform one initial source model into different (variable) target models for obtaining variable SPL members. In order to guide the generation of variable target models, we created three types of transformation rules: (1) base rules, (2) control rules, and (3) specific rules. However, these core assets are not enough to generate complete applications. Since some functional requirements are not generated during the automatic transformation processes, we manually complemented the applications in the application engineering process.

In this paper, we present the implementation of this approach using two representative frameworks based on the Model Driven Engineering (MDE) principles. We present a comparison of the features provided by the frameworks used to perform the implementation. These frameworks are the Graphical Modeling Framework (GMF) [6] that implements the OMG-MDA standard, and the Generic Model Environment (GME) [7] that implements the Model Integrated Computing (MIC) standard [8]. For building transformation rules, we use the ATLAS Transformation Language (ATL) [9] for the GMF implementation, and the Embedded Constraint Language (ECL) [10], that supports the concept of aspect model weaving combined with the idea of model driven transformation, for the GME implementation. We use as evaluation/comparison criteria the meta-metamodel (M3) facilities provided for the two frameworks, the functionality of the environments for creating metamodels and generating model editor plug-ins, the support for (meta)models composition, and the characteristics of the transformation languages for creating model-to-model and model-to-text transformation rules. Thus, this paper is mostly about comparison between the GMF and the GME frameworks. Comparisons such as described in this paper can be useful for finding out which framework to use in practical scenarios. This work is part of the AMPLE project [11]. The aim of AMPLE is to provide a SPL development methodology that offers improved modularization of variations, their holistic treatment across the software lifecycle and maintenance of their traceability during SPL evolution.

The paper is organized as follows. Section 2 describes related work. Section 3 presents a brief introduction to the frameworks and the transformation tools that we use for the two different implementations. Section 4 introduces the application context that allows us to illustrate the two implementations. Section 5 gives a summary of the approach for managing variability in MD-SPLs construction processes. It also presents the two different implementations describing the process followed to create the core assets and describes the comparison of the features provided by the frameworks. Finally, section 6 draws conclusions and some future work.

2 Related Work

There is not many works related to the comparison between GMF and GME. In [12], Bézivin et al. define an interoperability scheme to exchange models between the two frameworks building operational bridges. For doing this, the authors present a case study of a system developed by using GME, and the required transformations for creating the 'equivalent' (meta-)models in the Eclipse Modeling Framework (EMF), which is the core modeling framework used by GMF. Recently, the MDD-TIF07 workshop [13] remarked the importance of providing a forum for the implementers of modeling tools to meet and discuss common challenges and solution techniques. In such a context, [14][15] presented two different implementations that use the Atlas Model Management Architecture (that include ATL), and the GME respectively, for building the same proposed system. Those works allow observing the features of each framework separately. In this paper, we base our experimentation in the construction of a common product line, using the same development approach, and evaluating specific comparison criteria: meta-metamodel (M3) facilities, functionality of the environments for creating metamodels and generating model editor plug-ins, support for (meta)models composition, and characteristics of the transformation languages.

3 MDE Frameworks and Transformation Languages

In this section, we introduce GMF, GME, and the two model-to-model transformation languages we use: ATL in the context of GMF and ECL in the context of GME. GMF, GME and their transformation languages are built to resolve similar problems in the context of MDD, and both of them can be used to create MD-SPLs.

3.1 The Graphical Modeling Framework and the ATLAS Transformation Language

GMF provides a generative component for developing graphical editors using the Eclipse Modeling Framework (EMF) [16] and the Graphical Editing Framework (GEF) [17]. EMF is a modeling framework for building tools and other applications based on data models. EMF provides tools to produce a set of Java classes for the model that makes it possible to visualize and edit them. GEF allows developers to create graphical editors from an existing application model. Thus, with EMF (using the Ecore metamodel) and GEF, GMF proposes and supports a process for building Eclipse-based functionalities. The concept of a graphical definition model is the core of GMF. This model contains information related to the graphical elements that will appear in a GEF-based runtime environment.

Currently ATL is a standard Eclipse solution to define transformations, and it is one of the major components of the Model-to-Model Eclipse project. ATL is a mix of a declarative and an imperative language and is used to build transformations composed of rules that define how elements of a source model are transformed into elements of a target model. ATL supports the creation of two kinds of rules: (1) matched rules (declarative programming), and called rules (imperative programming). The matched rules constitute the core of ATL declarative transformations. The called rules provide imperative programming facilities; they have to be explicitly called to be executed.

3.2 The Generic Model Environment and the Embedded Constraint Language

GME is based on the MultiGraph Architecture (MGA), which is a part of MIC. GME is a toolkit for creating domain-specific modeling and program synthesis environments, which use MetaGME as their metamodel (M3) for building domain models. Configuration of domain models is accomplished through metamodels (M2) specifying the modeling paradigm (modeling language) of the application domain. The modeling paradigm contains all the syntactic, semantic, and presentation information regarding the domain. This defines the family of models that can be created using the resultant modeling environment. GME supports multiple *aspect* modeling. It allows metamodel composition for reusing and combining existing modeling languages and language concepts.

ECL is a procedural transformation language. ECL is implemented into a model transformation engine called the Constraint-Specification Aspect Weaver (C-SAW) [10]. ECL is an extension of the OMG's Object Constraint Language (OCL), it extends OCL by providing a series of operators for changing the structure or constraints of a model. ECL provides features such as collection, model navigation and a set of operators to support model aggregations, connections, and transformations. The structure of ECL modules includes strategies and aspects. A strategy defines a specific transformation procedure, and an aspect is a special strategy that serves as the entry point of a transformation. An ECL specification consists of many strategies, and a strategy can call other strategies. A strategy is applied by traversing a model and matching elements of the model that satisfy a predicate.

4 Application Context

As case study, we created an SPL for the Cupi2 project [18]. Cupi2 is a project of the software construction group of the University of Los Andes from Colombia that proposes a new approach to teach/learn computer programming. The approach of Cupi2 is problem based: the problems used as learning exercises are classified by levels. There are 18 levels and each one adds new concepts to teach. As an example scenario, we use level 7 that includes, among other topics, the algorithms to perform searches and orderings using collections. All the Cupi2 examples are stand-alone applications without complex non-functional requirements; they are developed using the same technological platform, in this case Java. All the examples are structured by three components: the kernel, the user interface, and the tests. The kernel component implements the concepts of the business logic. The user interface component implements visualization of the information and the interaction between the user and the kernel component.

The kernel concepts and their relationships can be functionally represented and manipulated by data structures of type Group. A Group always has a main element that groups the other elements of the kernel. For example, in a Music Store application, it should be a MusicStore Group that assembles Discs, and each Disc assembles Songs. All the kernel elements have a set of properties and these properties are possibly related to other elements of the kernel. Finally, each kernel element is responsible to make its information persistent.

The graphical user interfaces (GUI) use panels, lists, labels, images, and radio buttons, among others. All the GUI elements are grouped in views of different types. There are two types of views that are mandatory for any Cupi2 application: (1) the MainView, and (2) the ExtensionView. The MainView is in charge of communication between the kernel component and the other views and groups all the other views of the GUI. The ExtensionView contains buttons that the student can use to add new functionality as part of the exercise.

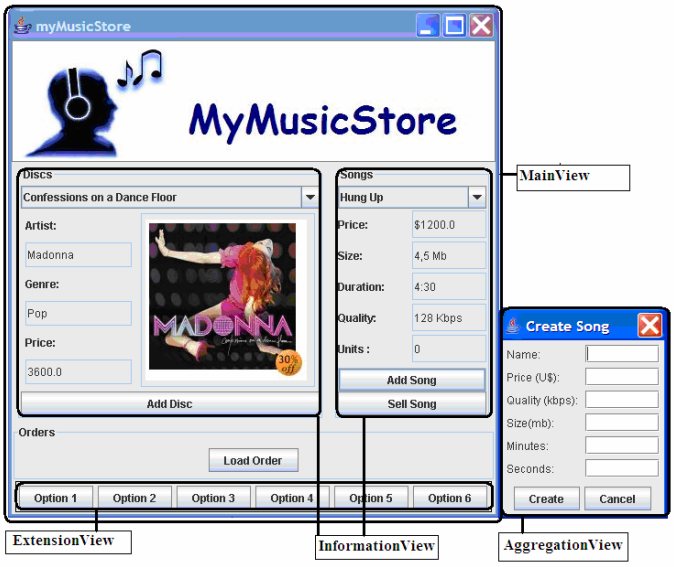


Fig. 1. A possible graphical user interface for a Music Store application

At the same time that we identify commonalities, we identify variability as well. The variability in the Cupi2 examples is related to the algorithms that manage the groups in the kernel component, and the presentation of the GUI. In the kernel component there are variable elements related to data structures, algorithms, and services to persist the different assemblies. The data structures that represent the groups can be fixed size or variable size containers, or can be linear structures like simple lists or doubly connected lists. Each type of data structure can be manipulated using different algorithms; for example, it is possible to manipulate a data structure with algorithms to make insertion, search, and ordering, among other services. Different implementations can be used to make information of the kernel elements persistent; for example by managing flat files, structured files or using object serialization. There is no a unique way to represent the kernel elements in the GUI. The user can select one or several types of views to represent the kernel elements. The types of view are: main, extension, set, search, information, and aggregation view.

The MainView assemblies other views and it is responsible for the communication between the kernel component and the user interface component. The SetView is in charge of presenting a set of grouping elements using components such as lists or tables. The SearchView is used to enter parameters to search in groupings. The InformationView is used to show the particular information associated to the attributes

of a kernel element of the problem, including images. The AggregationView is used to enter particular information associated to the attributes of new kernel elements. Fig. 1 presents an example of a possible GUI for the Music Store example showing four types of views.

The Cupi2 project is relevant for the comparison of GME and GMF since it needs the development of a SPL with all the traditional elements that it implies; for example, to develop a domain engineering and application engineering process, and to create (core-)assets in the problem and the solution spaces. In addition, the Cupi2 product line is not biased towards GME or GMF since it has business requirements and general problems related to any SPL, even when there is not non-functional requirements as concurrency, or real-time execution.

5 The Variability Management Approach

In this section, we present a summary of the approach presented in [6] for managing variability in MD-SPLs construction processes and we describe how the approach was implemented using GMF/ATL and GME/ECL. We separate concepts related to a SPL in domains to manage variability. These domains are the business logic (high abstraction level), the architecture domain, and the technological platform domain (low abstraction level). Then, for each domain, we create a metamodel.

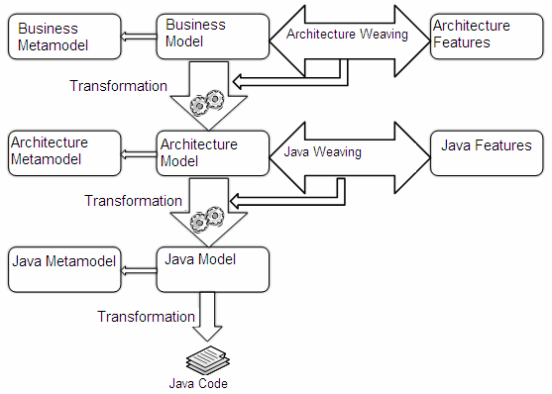


Fig. 2. Product line member creation.

Our strategy follows the MDA approach and it is based on the automatic transformation (refinements) of models until obtaining executable applications. Thus, for creation product line members (see Fig. 2), we start from a business logic model, we transform it into an architectural model, after that we refine it into a model in the technological platform domain and finally we generate source code from it. It is important to note that each transformation process is guided by a feature selection done by the product line developer. To achieve this, we build as core assets (1) metamodels for each domain (Fig. 3-first column), (2) feature models for each target domain (Fig. 3-second column), (3) mapping models (Fig. 3-third column), and (4) three types of transformation rules (Fig. 3-fourth column).

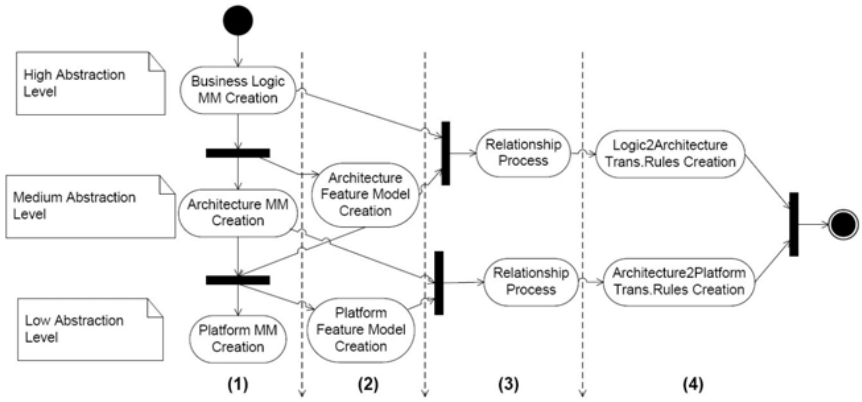
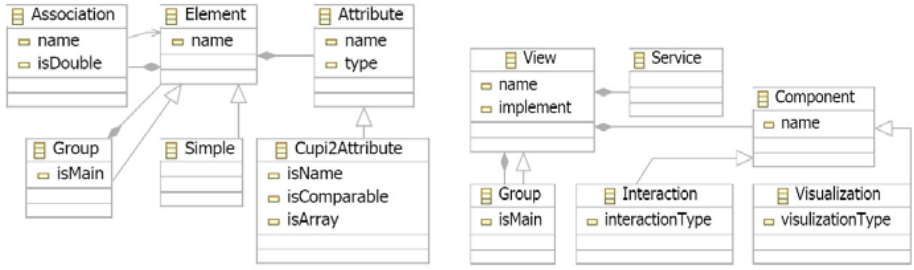


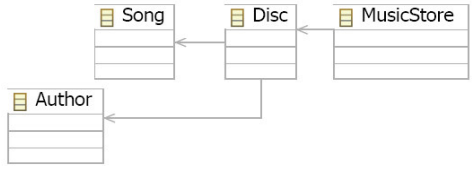
Fig. 3. Domain-Engineering process.

5.1 Metamodels Definition

For the creation of the Cupi2 (Level 7) SPL we defined three different metamodels as core assets: business logic, architecture, and technological platform metamodel. Business logic metamodel includes the essential concepts of the problem; architecture metamodel includes concepts of the architecture of the applications including the user interface; finally, the technological platform metamodel includes concepts of the language as for example packages, classes, methods and attributes. For the last one, we created a simplified Java metamodel.



(a) Business logic metamodel (b) GUI architecture metamodel



(c) Example of a business logic model

Fig. 4. Business logic and GUI architecture metamodels

The business logic metamodel has concepts to describe a set of elements related between them using Group structures. Each element can have a set of

Attributes (see Fig. 4a). The architecture metamodel includes design concepts of the Cupi2 examples. The business logic architecture metamodel introduces the concept of Service. Services are needed to manipulate the data structures of the examples. The GUI architecture metamodel includes the concepts of graphical and interaction elements that will be part of the GUI of generated applications (see Fig. 4b). Fig. 4c presents an example of a model that conforms to the business logic metamodel. The MusicStore and Disc elements conforms to the Group meta-concept, and the Song and Author elements conform to the Simple meta-concept. For the metamodel definition activity, we focus on the meta-metamodel (M3) concept facilities, the environments for creating metamodels, and the generated model editor plug-ins.

5.1.1 Metamodel Definition in GMF

For the metamodel definition, GMF is based on the Eclipse Modeling Framework (EMF), which is a Java implementation of a core subset of OMG-MOF called Ecore. Using GMF, we created EMF models using the Eclipse Ecore diagram editor, which allows us to create models similar to UML class diagrams. The structural organization of Ecore (M3) includes meta-concepts like `EPackage`, `EClass`, `EAttribute`, `EReference`, `EOperation`, `EDataType`, `EEnum`, and `EEnumLiteral` among others. The Ecore metamodel admits multiple inheritance, therefore one meta-class can have different super meta-classes; and the `EReference` concept is used to define association or aggregation relationships between classes. Using these concepts, we built one metamodel for each domain. We created the plug-in editors to edit models conform to the EMF metamodels using the GMF framework for building Eclipse-based functionalities. This framework separates the definition of the domain model (M2), the graphical model associated (concrete syntax), and the tools to manipulate domain concepts with the associated concrete syntax. Thus, the domain model is not coupled with concrete syntax, and different concrete syntax can be associated. Finally, the generated Eclipse plug-in editors are Java plug-ins; thus, we can modify these to include, for example, constraints associated to the domain model.

5.1.2 Metamodel Definition in GME

MetaGME is the meta-metamodel (M3) for the GME framework. MetaGME is a UML profile with meta-concepts such as `Project`, `Folder`, `Model`, `Atom`, `Attribute`, `Connection`, `Proxy`, `Reference`, and `Aspect` among others. A `Project` contains `Folders`; a `Folder` contains `Models`. A `Model` is composed of `First Class Objects` (FCO). `Relationships` between `FCO` are `Connections` or `References`. A `Connection` can only express relationships between objects contained by the same `Model`. `References` and `Proxies` are useful to express a relationship between objects contained by different models (but contained by the same `Project`). The `Proxies` are references pointing to concepts defined elsewhere in a metamodel. Finally, `Aspects` provide visibility control; they define points of view on models. Using the MetaGME concepts, we built one metamodel for each domain. However, we reused concepts of different metamodels using `proxies`. Thus, we created `FCOs` (`Models`, `Atoms`, and `Connections`) in the business logic metamodel, and we reused them in the architectural metamodel. For example, we

reused the Element concept (Model) and the name concept (Attribute) defined in the business logic metamodel, in the logic architecture metamodel (Fig. 4b). Thus, we created metamodels using (composing) concepts of other metamodels, and adding information to refine the concepts when it was needed. We used metamodels to generate the domain-specific environment (model editors). We created one Aspect for each metamodel and related one icon to each FCO, thus each icon represents the concrete syntax associated to each metamodel concept. Finally, the generated domain-specific environments are represented by XMI code that is interpreted by the GME framework.

5.1.3 Metamodel Definition Comparison

In [12], there is a proposal of an interoperability scheme to exchange models that conform to the Ecore metamodel and the MetaGME metamodel. However, there is not a one-to-one mapping between the concepts of each environment.

Both meta-metamodels offer concepts to organize elements, *i.e.*, EPackage in Ecore or Folder in GME. They also offer concepts to represent domain concepts, EClass and EAttribute in Ecore, and Model, Atom and Attributes in MetaGME. EClass is the only concept in Ecore to define domain concepts, while in MetaGME Models are complemented with Atoms. The Atoms are specialized Models which can not group others Models or Atoms. Thus, Models aggregate concepts, and Atoms are always "aggregated" concepts. One EAttribute is related just with one EAttribute in Ecore, but in MetaGME one Attribute can be reused to be related with different Models or Atoms. Regarding the way of associating domain elements, in Ecore the EReference concept is used to define association or aggregation relationships between EClasses. In MetaGME Relationships between domain concepts can be Connections or References. Using Connections is possible to relate objects included in the same Model. Using References is possible to express a relationship between objects contained by different models. The Ecore metamodel does not define equivalent concepts for MetaGME Proxies, Aspects, and Constraint concepts, and does not allow the association of graphical information with the metamodels. The MetaGME metamodel does not define equivalent concepts for the Ecore EProperties concept.

For us the Ecore metamodel looks essentially as an UML class diagram, and the MetaGME metamodel has extensions for composing metamodels using Proxies, Aspects, different types of relationships and inheritance, admitting the reuse of Attributes and the creation of constraints directly associated with metamodel concepts. We found that the main problem with metamodel composition using GME is that metamodels can be *low cohesive; by* reusing concepts of different domains, or domains concepts of different abstraction level.

Regarding the environments to create metamodels, both environments provide similar graphical facilities. The MetaGME offers the option to associate graphical information to domain concepts. GMF complements the EMF facilities by also offering facilities to define graphical information. Finally, once the metamodels are created, by using both frameworks it is possible to generate graphical environments to create models that conform to the defined metamodels. We found the GMF plug-in process creation more complex than the GME editing model creation. However, the

model editor plug-ins generated by GMF are easily modified/extended since they are built using Java source code; while the GME environments for editing models are defined using XML code that is interpreted by the GME framework, which is more complex to modify/extend.

5.2 Feature Models Definition

We created feature models based on the characteristics of the applications in the architecture and the technological platform domains. To define our feature models, we used concepts such as feature nodes, feature groups, and cardinality.

Fig. 5a presents a part of the feature model for the architecture domain. These features are related to the commonalities and variability identified in section 2 of this paper. Fig. 5b presents feature nodes of the technological platform domain (Java) feature model. The relationship between the features nodes implies that nodes selected in the architectural feature model constrained the selection the user can do in the platform feature model. For example, selection of PrintWriter or BufferReader features (Fig. 5b) can be done only if the Files feature (Fig. 5a) is selected before.

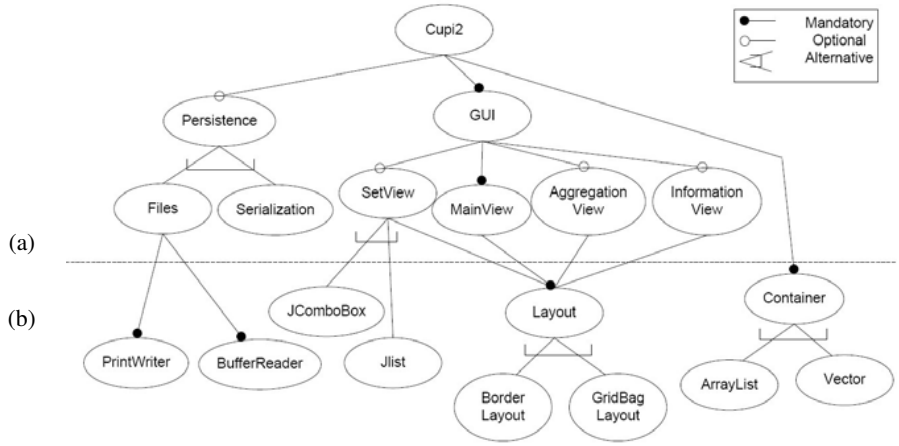


Fig. 5. Feature model for Cupi2 domains

5.3 *Mapping* Models and Transformations Rules Definition

Users define their preferences by means of the feature models; they have to make selections on features before the initialization of the transformation processes. The selection is done by associating (source) model elements and (target) feature nodes. Thus, for example for the music store application, the classes Disc and Song, both conform to the Element concept (Fig. 4a); one user may want to design the persistence for Disc elements using files, and the persistence for Song elements using serialization. For this, the user must relate the Disc element with the Files feature node (Fig. 5), and the Song element with the Serialization feature node.

Transformation rules are defined in terms of metamodel concepts. It implies that a same source model is always transformed into the same target model. Since we need

to transform the same metamodel concept into different target concepts according to the features related, we need to create different rules. For example, for the same music store application, even when the Disc and Song elements conform to the Element concept, different transformation rules must be created to transform Elements to obtain the Files feature or the Serialization feature.

For doing this, we created three types of transformation rules. These types of rules are: (1) base rules, (2) control rules, and (3) specific rules. In order to guide the creation and posterior execution of these transformation rules, we created new composed models containing relationships between metamodel concepts and feature nodes, and new composed models containing relationships between model elements and feature nodes. We call the first *meta-mapping* models, and the second *mapping models.*

The meta-mapping models link metamodel concepts to feature nodes generating a new composed model. Each link means that one source metamodel concept can be transformed to obtain the target feature node. Thus, these meta-mapping models are core assets used as design elements for constructing the different transformation rules. Using the meta-mapping models, we can identify (1) the metamodel concepts that have to be always transformed into the same target feature, and, (2) the metamodel concepts that can be transformed into different (variable) target features. For the first case, we create base rules; and for the second, we create control and specific rules. Thus, the base rules allow us for the generation of the commonalities of the product line, and the control and specific rules the variability. Fig. 6 presents a meta-mapping model between the business logic metamodel and the feature model of the architecture domain. In Fig. 6, the link between Group and the mandatory feature MainView indicates that a MainView is always created for Group elements (isMain == True). For this connection, we create a base rule. The links between Element, and the Serialization and Files feature nodes indicate that an element of type Element can implement its persistence using Files or Serialization methods. For these connections, we created one control rule and two specific rules.

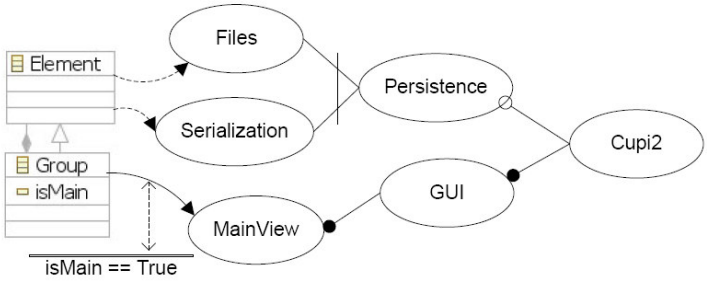


Fig. 6. Meta-Mapping model between the business logic metamodel and the architecture features model

We link model elements with feature nodes for creating new composed mapping models in the application engineering process. These models are created when the user selects model elements and relates them to selected features to guide the generation process. Since these mapping models are inputs to the transformation processes, it is mandatory to use a mapping metamodel as core asset in the

domain-engineering process. We use the ATLAS Model Weaver (AMW) [19] metamodel for the GMF implementation, and we create one mapping metamodel for the GME implementation.
For this activity we focus on the (meta)model composition facilities provided by the two frameworks needed to create mapping models, and the transformation language facilities for creating our three types of transformation rules.

5.3.1 Mapping Models and Transformation Rules Definition in GMF

For the GMF implementation we create the (meta)mapping models using the AMW metamodel and its plug-in. The AMW is integrated into Eclipse for establishing relationships (*i.e.*, links) between models. The created links are stored in a new model, called weaving model that conforms to a weaving metamodel. We create the transformation rules using the declarative and imperative facilities provided by ATL. We implement the base rules using declarative programming, and the specific rules using imperative programming. The control rules are implemented in a mixed way, it means, they have a declarative section and an imperative section. The ATL modules we created have as input the source model that will be transformed, and a mapping model. The mapping model is used, in the control rules, to decide which transformation rule must be executed according to the feature linked to the element that is being transformed. Listing 1 presents an example of control and specific rules. The `setView` is a control rule. The declarative section is in the lines 2 to 4 and the imperative section is in the line 5. The `setView` rule searches in the mapping model the elements that have the feature `SetView` associated (line 2). For those elements a `View` is created (line 4) and the `addComponent` imperative rule is called (line 5). The `addComponent` rule creates concepts associated to the created `View` giving the characteristics needed for a `SetView`. The `addComponent` rule is a specific rule which has the imperative sentences in the line 9. When this rule is called from the control rule, a `Visualization Component` is created (line 8), the parameter type is assigned to this component, in this case `List` (line 8), and the created component is added to the previous created `View` (line 9).

```
(1)     rule setView{
(2)       from link: mapping!Link(link.feature=='SetView')
(3)         using{ element: kernel!Element = link.getElement(); }
(4)       to   target: Architecture!View()
(5)         do{ addComponent(target, #list);       }
(6)     }
(7)     rule addComponent(view: View, t: Type){
(8)       to component: Architecture!Visualization( type<-t )
(9)       do{ view.attribute.add(component); }
(10)    }
```

Listing 1. ATL control and specific transformation rules

5.3.2 *Mapping* Models and Transformation Rules Definition in GME

We create meta-mapping models without tool support because using GME it is not possible to link metamodels concepts (M2) and model elements (M1). To have a mapping metamodel, we use the `Reference` and `Proxy` MetaGME concepts for composing models. `References` are similar to the pointer concept found in various programming languages. Thus, the metamodel includes `Proxies` of all the concepts

that can be linked with Feature Nodes, and a common *"super-type"* `Model` concept (MetamodelElement in Fig. 7) for all of them. This concept has a `Connection` (aggregation) with one `Reference` concept that refers the FeatureNode concept of the feature metamodel. Thus, the model elements can be related through `References` with different FeatureNodes in the application engineering process.

Fig. 7 presents the GME mapping metamodel.

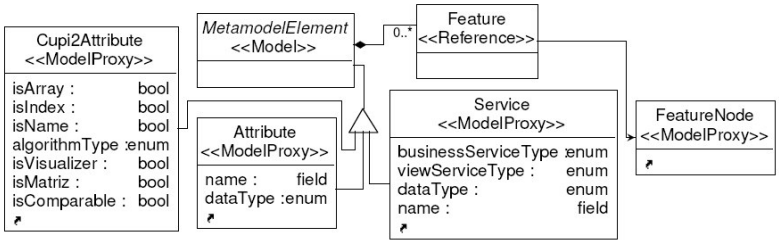


Fig. 7. GME mapping metamodel

Once we created the meta-mapping models, we built transformation rules using ECL. Transformation rules in ECL are written as `Aspects` and `Strategies`. Thus, we define base strategies to create the commonalities, specific strategies to create variability characteristics, and control strategies to decide which specific strategy is used to create variable functionality.

```
(1)     strategy viewArch(name : String){
(2)       viewArchM := rootFolder().addModel ("ViewArch", name);
(3)       mainView := viewArchM.addModel("Group","MainView");
(4)       elems := this.models()->select (m|m.kindOf()=="Group" or
            m.kindOf()=="Simple");
(5)       elems->transElem(mainView);
(6)     }
(7)     strategy transElem(mainView: model){
(8)       featureNodes := self.modelRefs("Features");
(9)       featureNodes->select (m|m.name()=="SetView")-> transElemSetView(this,
            mainView);
(10)    }
(11)    strategy transElemSetView(elem: model; mainView: model){
(12)      setView := mainView.addModel("Group", elem.name());
(13)      setView.AddViewComp(setView,"List", "List", "Interaction");
(14)      elem.models("Cupi2Attribute")->select(m|m.getAttribute("isComparable"))
            ->AddButtToManageList(setView);
(15)    }
```

Listing 2. ECL base, control, and specific strategies

Listing 2 presents an example of base, control and specific strategies. The `viewArch` strategy is a base strategy to transform business logic models to GUI architecture models. In line 2 an element that conforms to `ViewArch` is created. This is the root of any `ViewArch` model. As part of the transformation logic, the `mainView` is always created (line 3). Finally a control strategy is called (line 5) to transform elements that match a condition defined in the line 4. The `transElem` control strategy (line 7) explores business logic models searching `References` from `self`, which is a `Group` or a `Simple` concept, to different feature nodes. For this example the `References` to a `SetView` feature node are searched (line 9). For each reference found the `transElemSetView` strategy is called (line 9). Finally, the `transElemSetView`

specific strategy (line 11) transforms items that conform to the `Element` concept to `SetView` features. This means that the rule creates target elements that allow having a GUI with a `SetView` (lines 12-14).

5.3.3 *Mapping* Models and Transformation Rules Comparison

The meta-mapping models and mapping models creation are model composition activities. In our approach, model composition appears in three contexts: (1) composition of metamodels, (2) composition of metamodels with models, and (3) composition of models. The two first are done in the domain engineering process. The first takes place in the metamodel definition activity (section 4.1) and the second, in the meta-mapping models creation. The third takes place in the mapping models creation in the application engineering process.

The Ecore metamodel in the contexts of GMF does not support the three composition activities. However, it is possible to do the same using AMW. We found that using AMW allows for composing models using a graphical interface to relate concepts of the source models, maintaining *cohesive* each one, and creating new weaving models with new domain specific targets.

In the context of GME, the MetaGME metamodel supports the composition of metamodels; however, the composition of metamodels with models is not supported, and the composition of models is supported using concepts provided by MetaGME to create a composition metamodel and thus create composed models. This is the case of the mapping metamodel that we created. It is important to remark that models can be composed just with other models in the same `Project`.

In the construction of transformation rules, we found ECL an intuitive language to define transformation rules. Furthermore, ECL supports the concept of aspect model weaving. Note that composition of models using ECL implies that the source models and the target model conform to the same metamodel. We found that even while the ECL language has sufficient number of operators for transforming/composing models, it does not support definition of declarative strategies, therefore implying that we always need to define the logic to traverse across the models. Thus, strategies for transforming models with ECL imply writing programs that navigate manually through the models that will be transformed or composed. Another thing is that ECL has some problems manipulating `boolean` type and control structures. The `boolean` type does not have symbolic representation for `boolean` elements defined in GME models, and `boolean` elements can not be defined in ECL programs. Furthermore, the control structures do not admit using else-if conditions, which is useful to avoid problems with exclusive decisions, and do the evaluation of grouped conditions more elegant.

Different to ECL, ATL facilitates the creation of imperative and declarative rules. Furthermore using ATL it is also possible to build rules that trigger the execution of specific rules. This is the case of the control rules. These can be used, as in ECL, to build composed models; besides, using ATL source models and the composed target models can conform to different metamodels. Finally, different to ECL at the moment, there are various ATL discussion groups, examples and available resources that support work with ATL.

5.4 Models-to-Text Transformations Rules Creation

We use Acceleo [20] as model to text transformation language in the GMF implementation. This tool is specialized in the generation of text files (code, XML, documentation) starting from models. Using Acceleo we take the EMF models and use templates for the source code generation.

For the GME implementation, we have to create our own Java generator to navigate the (XML) GME models and generate Java code.

The Model-to-Text (Java) transformation facility is less supported in GME. Even when there are available technologies to access GME data, we found it restricted and poorly supported. GME components can be built using the Unified Data Model module (UDM) [21]. However, these support tools provide restricted languages to manipulate models that do not allow the use of control mechanisms as loops and conditional statements, and the traceability management between models and the generated text.

Different to this, there are various versions of tools that support the model-to-text transformation construction process in the context of GMF. Examples of these are Acceleo and MOFScript [22]. These tools allow for generation of implementation code or documentation from models, generation of text from MOF-based models, manipulate control mechanisms as loops and conditional statements, and managing traceability between models and generated text. One important thing is the support for these tools. For example, currently the MOFScript language is a candidate in the OMG-RFP process on MOF Model-to-Text Transformation.

6 Conclusions and Future Work

In this paper, we presented the implementation of an MDA approach for managing variability in product line construction using the GMF and the GME frameworks; and the comparison of the features provided by the frameworks for these implementations. The approach allows managing variability at *model level* for MD-SPLs construction. This means that we enlarge the SPL scope by making possible to generate different (variable) SPL members starting from the same initial business logic (platform independent) model, guided by a specific features selection. The implementation of this approach using two of the most representative frameworks and the followed comparison allows us to know each framework and take decisions about which of them is better for specific experimentations in the MD-SPLs field.

Currently we continue working on the presented approach focusing in (1) the constraint management for delimiting the SPL scope, (2) the inclusion of information of functional requirements in the metamodels, and (3) the traceability management required for maintaining core assets and generated applications. For this, we have chosen GMF for future experimentation. We found that GMF allow us to create cohesive metamodels, compose and manipulate these using the AMW, optimize the transformations using a declarative language and its optimized engine, create transformation rules for transforming the same models using variable transformation logic, and transform models to source code supported by tools with necessary characteristics. In addition, we found the current eclipse community support to be a valuable addition.

References

1. Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
2. Kleppe, A., Warmer, J., Bast, W. MDA Explained: The Model Driven Architecture – Practice and Promise. Reading, Addison-Wesley (2003)
3. Santos, A.L., Koskimies, K., A. Lopes, A.: A Model-Driven Approach to Variability Management in Product-Line Engineering. In Nordic Journal of Computing, Volume 13, Issue 3 (September 2006) 196-213
4. Wagelaar, D.: Context-Driven Model Refinement. In Proceedings of the Model Driven Architecture: Foundations and Applications Workshop, Linköping, Sweden (June 2004) 189-203
5. Garces, K., Parra, C., Arboleda, H., Yie, A., Casallas, R.: Administración de Variabilidad en una Línea de Producto Basada en Modelos. In Proceedings of the Congreso Colombiano de Computación, Bogotá, Colombia (April 2007)
6. GMF, Graphical Modeling Framework site. On line: http://www.eclipse.org/gmf/
7. Davis, J.: GME: The Generic Modeling Environment. In Proceedings of the 18th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, USA (October 2003) 26-30
8. Ledeczi A., Balogh G., Molnar Z., Volgyesi P., Maroti M.: Model Integrated Computing in the Large. In Proceedings of the IEEE Aerospace Conference, Big Sky, USA (March 2005) CD-ROM
9. Jouault, F., Kurtev, I.: Transforming Models with ATL. In Proceedings of the Model Transformations in Practice Workshop, Montego Bay, Jamaica. (October 2005) 128-138
10. Gray, F., Bapty, T., Neema, S., Schmidt, D., Gokhale, A., Natarajan, B.: An approach for supporting aspect-oriented domain modeling. In Proceedings of the 2nd international conference on Generative programming and component engineering, Erfurt, Germany (September 2003) 151-168
11. Bézivin, J. Brunette, C. Chevrel, R. and Jouault, F., Kurtev, I.: Bridging the Generic Modeling Environment (GME) and the Eclipse Modeling Framework (EMF). In Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development, San Diego, USA (October 2005)
12. AMPLE Project site. On line: http://ample.holos.pt/pageview.aspx?pageid=1&langid=1
13. TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum. Zurich, Switzerland (June 2007)
14. Barbero, M., Bézivin, J., Jouault, F. Building a DSL for Interactive TV Applications with AMMA. In Proceedings of the TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum. Zurich, Switzerland (June 2007)
15. Molnár, Z., Balasubramanian, D., Lédeczi, A. An Introduction to the Generic Modeling Environment. In Proceedings of the TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum. Zurich, Switzerland (June 2007)
16. EMF, Eclipse Modeling Framework site. On line: http://www.eclipse.org/emf/
17. GEF, Graphical Editing Framework. On line: http://www.eclipse.org/gef/
18. Cupi2 Project site. On line: http://cupi2.uniandes.edu.co.
19. Didonet Del Fabro, M., Jouault, F.: Model Transformation and Weaving in the AMMA Platform. In Proceedings of the Generative and Transformational Techniques in Software Engineering Workshop, Braga, Portugal (July 2005) 71-77.
20. Acceleo site. On line: http://www.acceleo.org
21. Magyari, E., Bakay, A., Lang, A., Paka, T., Vizhanyo,A., Agrawal, A., Karsai, G.: UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages. In Proceedings of the OOPSLA Workshop on Domain-Specific Modeling, Anahiem, USA (October 2003)
22. MOFScript site. On line: http://www.eclipse.org/gmt/mofscript/

A formal approach to the cross-language version management of models

Harald Störrle

Institut für Informatik, Universität Innsbruck
Technikerstr. 21a
6020 Innsbruck, Austria
Harald.Stoerrle@uibk.ac.at

Abstract. This paper presents an approach to formally define version-management related operations and properties for models irrespective of the modeling language, the underlying data structure, tool or other technical constraints. The approach transforms models into an extremely simple format defined both as a meta-metamodel and mathematically. The operations necessary for version-management tasks are also defined formally. The approach has been implemented in Prolog and tested successfully in two large scale industrial projects. An earlier version of the first part of this paper has been published as [22].

1 Introduction

1.1 Motivation

Using models is considered to be an industry best practice for a great many situations, including the early phases of large software development projects, schema integration for databases, and business process management and optimization. Very often, large families of models are created in such settings and many of the models may have a prolonged lifetime with numerous changes, additions, and updates. Thus, versioning of models is imperative for real life projects. Still, it has so far been neglected.

Today's approaches to model versioning are very restricted. They might cover the UML modeling language, consider only one model type (such as static structure models, aka. "class models"), and build heavily on the UML meta model as the data structure and XMI as the file format, and rely on the existence of persistent globally unique object identifiers for model elements. Obviously, such an approach is rather limited by the parameters mentioned – can problems and experiences with this approach be generalized to (all) other parameter settings? Does this configuration of parameters exhibit all phenomena to be found in model versioning, in general? It is not clear, how such questions could be answered. It is clear, however, that there are many such parameter settings in real life projects.

In this paper, we try to circumvent these questions by creating a least common conceptual domain for expressing models irrespective of the language, tool,

format and so on. That is, it should be possible to transform all conceivable models into this common conceptual domain without loosing information relevant to versioning, define problems, phenomena, and algorithms for versioning purposes, and study them *in general* by just studying a single one approach, rather than having to study each and every specific setting.

So, in a sense, we strive for a "sound and complete" approach, metaphorically speaking, where completeness would mean that all phenomena specific to modeling that occur in any setting will occur in this setting. Conversely, soundness would mean that all phenomena occurring in the approach presented here would occur in some real setting.

1.2 The problem space

In general, there are (at least) three dimensions of models that are relevant for versioning:

modeling language the notation or family of notations in which a model is described (including, of course, textual as well as visual notations);
model type the family of concepts used for modeling together with their properties and relationships;
data structure the data structure in which a model is stored in memory, or the file format in which a model is stored.

Concerning the modeling language, although UML is claimed to be the "*lingua franca of software engineering*", there are still many other modeling languages in widespread use today. For instance,

- for embedded/real time systems modeling (e. g. in the telecom industry), SDL/MSCs are the most common modeling notation;
- for business process management and modeling (e. g. in the standard software and customizing businesses), ARIS/EPCs are the most common modeling notation;
- for corporate data models, database design, and ontology modeling, ER-like models are still more common as a modeling notation than UML.

Even when focusing on the UML as the modeling language for a difference algorithm, there are 13 different diagram types defined in the UML, all of which have several types of usage and specialisation constituting an individual model type each. And then, there are dozens of other modeling language families with their own special kinds of models.

Furthermore, even when also restricting ourselves to class diagrams, there are different UML-versions of class diagrams to choose from like [25, 17], different XMI-versions, a multitude of tools with their own interpretations of UML and XMI, their own bugs and patches and so on.

That is, even in the limited scenario we have mentioned at first, there are still many variation points, each of which may have a significant influence on a

difference algorithm, resulting in false positives or false negatives when comparing models created with, for instance, different versions of the same tool. Only when all these parameters are fixed do we have a sufficiently precise foundation for the definition of a difference algorithm, and thus versioning.

That means, on the other hand, that almost all combinations of modeling languages, model types, and data structures are actually incompatible, even if there are conceptual similarities that should be found by a difference algorithm. To some degree, this problem may be circumvented by automated model transformations into, say, a certain combination of UML and XMI versions of a certain tool. This does not work for all cases, though, for instance, many text-based models such as the popular use case tables have not counterpart in UML. Similarly, ordering of use cases in use case maps (cf. [4]) is not part of the UML. The same problem arises even more stringently for Domain Specific Languages.

1.3 Related work

There are several approaches to define differences on UML class models for a specific XMI representation (see [24] for a comparison, and in general [12]).

The most restricted approach is to record the model edit history of a tool and use this information to determine the difference between a base model and a model derived from it. Such an approach is not just restricted to the modeling language, data structure, version etc., but also to the tool, and can hardly be used for industrial scale projects.

Most approaches are restricted to a single type of model, typically UML 1.x class models represented in some version of XMI (see [17, 25]). For instance, Fujaba 4.3 accepts only class models conforming to UML 1.3 and XMI 1.2. Kelter et al. have proposed an approach that deals with all model types of UML 1.x, but is still restricted to a particular data structure (some XMI-version). More general approaches are put forward in [2, 11, 1] which define versioning operations based on the meta-metamodel of UML (i. e. MOF).[1]

All of the previously mentioned approaches have rather tight limits on the class of modeling languages, data structures and so on. Another approach that overcomes these restrictions is Rondo (cf. [13, 14, 3]). It defines a kind of relational algebra for general XML-structures. While this avoids the restrictions of UML-centered approaches, it retains the restriction to the XML (meta-)format. In particular, dealing with the full complexity of arbitrary XML is not necessary, as most models are "flat" rather than tree-like. Another drawback is the lack of mathematical foundation. Having a versioning tool is necessary for practical purposes, but it would be preferable to have a formal basis in order to define those

[1] Hein and Ritter [11] deal with versioning across different tools. Ahrens [1] examines model differences and model merges based on meta models. Both of these references appear in [12] which is currently the most up-to-date and comprehensive collection of the state of the art in model versioning. Unfortunately, some of the contributions to [12] are in German rather than English, in particular, [11, 1]. We apologize to all non-German speaking readers.

operations, properties, and phenomena precisely that occur during versioning and merging.

The SHORE system (see [15]) is an approach to representing and querying models using post-relational database technology. SHORE was designed to store software design documents in an XML database and used Prolog as a query language.

Gruhn and Laue [10] present an approach where Prolog is used to encode EPCs (in a rather ad hoc way). This representation is then used to check consistency conditions on models. This approach is limited to EPCs, unfortunately.

The meta-metamodel we will present below is somewhat reminiscent of Mario Bunge's ontology (see [5, 6]) or the Resource Description Format (RDF, see [20]).

1.4 Approach

In this paper, we propose a two-step approach to difference computations and similar operations. In the first step, models from all sources are transformed into a common meta-metamodel for differencing. This way of describing abstract structures has proved to be quite effective in industrial settings since it appeals very much to practitioners.

In the second step, this meta-metamodel is then formalised so that version-management operators may be defined, and the phenomena that occur may be studied rigorously. Finally, we provide tool support for case studies in order to investigate the potential for practical applicability of our approach. The algorithms used in our approach are more or less naive.

2 A least common conceptual domain for models

In our approach, a model is first transformed into an efficient representation according to the Meta-Metamodel shown in Figure 1. As differencing is the focus of this paper it has been called the *difference* meta-metamodel, though it may be used for many other purposes as well, including model queries (cf. [23]), transformations, or as a kind of interlingua to translate between different modeling languages. It should be obvious, that it is possible to transform any metamodel into this simple data structure. As an example, we will sketch the mapping for UML and ARIS/EPC into this meta-metamodel in Section 2.2.

2.1 Formal foundations

Now we formalize the difference meta-metamodel by a set of simple mathematical domains. We define more or less one domain per metaclass. The goal of this formalization is to create a frame of reference to precisely describe the phenomena occurring and the operations to be executed when versioning models.

Identifier An identifier is an arbitrary symbol. The domain of identifiers is represented by $\mathcal{I}$. All identifiers are preceeded by the special symbol # to set them apart from other symbols.

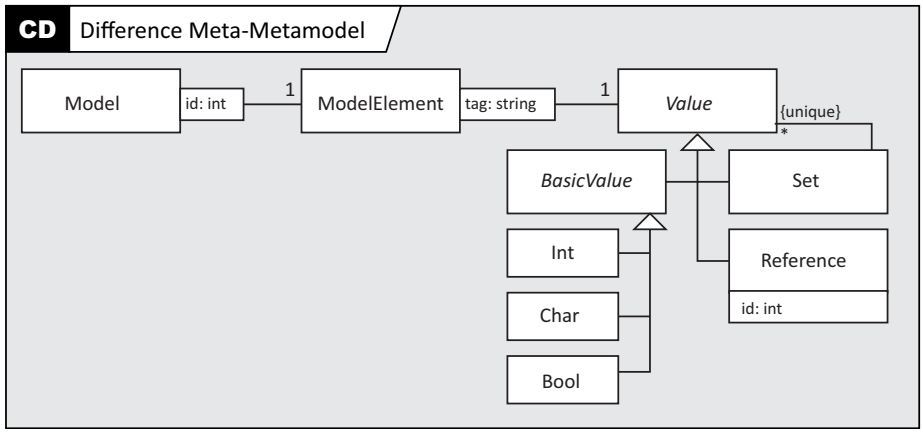


Fig. 1. A Meta-metamodel as a normal form for arbitrary modeling languages.

Tag A tag is an arbitrary symbol, the tags domain is represented by $\mathcal{T}$.

BasicValue A basic value is any of the usually available basic data types such as integers, characters, boolean values and so on. We will assume that values are dynamically typed such that the string "1", the character '1' and the integer 1 are three different values. The domain of basic values is represented by $\mathcal{V}$ defined as $\mathcal{V} = Int, Char, Bool, \ldots$.

Reference A reference is an identifier referring to a model element. The domain of references is represented by $\mathcal{R}$ which is identical to the domain $\mathcal{I}$.

Set A set is a collection of values. The domain of basic values is represented by $\mathcal{S}$ defined as $\mathcal{S} = \mathcal{P}(\mathcal{V})$.

Value A value is either a basic value, a set or a list. The domain of basic values is represented by $\mathcal{V}$ defined as $\mathcal{V} = \mathcal{B} \cup \mathcal{S} \cup \mathcal{L}$.

Model element A model element is a mapping from identifiers to values. The domain of model elements is represented by $\mathcal{E}$ defined as $\mathcal{E} = \mathcal{T} \to \mathcal{V}$.

Model A model is a mapping from identifiers to model elements. The domain of models is represented by $\mathcal{M}$ defined as $\mathcal{M} = \mathcal{I} \to \mathcal{E}$.

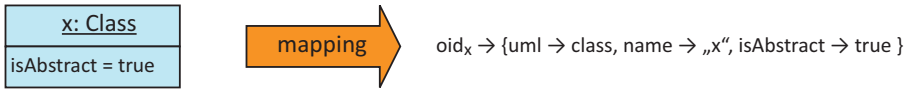
This metamodel and the domains are simple enough for a straightforward formalization, yet expressive enough to define all operations relevant for model versioning (difference, matching, merging), which we will define in the next section. Many other operations beyond versioning may also be expressed in this approach, but are not the focus of this paper.

2.2 Mapping from real-life models to formal domains

UML and ARIS/EPC (Event Process Chains, frequently used in the variety of extended EPCs, eEPCs) are two of the most widely used modeling languages

today (see [18, 21] for UML and [8] for ARIS/EPCs). They are rather different in terms of content, tool support, and degree of formalisation so that we are able to demonstrate the versatility of the framework defined above.

For UML, the mapping into the domains defined above is simple indeed, since UML comes with a metamodel. The set $\mathcal{T}$ of tags is exactly the set of all meta-attributes in the UML metamodel plus the additional tag uml to hold the type of the metaclass. The set $\mathcal{I}$ is the set of all identifiers of instances of UML metaclasses in a given model m. For every instance $\mathbf{x}$ of a UML metaclass $\mathbf{c}$ in model $\mathbf{m}$, we create a tuple $i \mapsto e$, where i is the identifier of $\mathbf{x}$, and e is the attribute set of $\mathbf{x}$. The attribute set of $\mathbf{x}$ is a function from the set of attributes of the metaclass $\mathbf{c}$ and all its superclasses (that is, all elements of the transitive closure of the generalization relationship) to their respective values. In other words, the attribute set of $\mathbf{x}$ is the set of elements $t \mapsto v$, where t is a meta-attribute of $\mathbf{c}$ and v is the value of $\mathbf{z}$ in $\mathbf{m}$. Observe that there is an exception to this rule: metaclass $\mathbf{c}$ is mapped into an attribute as well. The whole mapping is illustrated by Figure 2. This mapping schema may be applied to all languages that have a metamodel comparable to the UML metamodel, including, of course, all MOF-compliant metamodels.



domain	UML mapping for a given model $\mathbf{m}$
$\mathcal{I}$	the object-identifiers of all instances of concrete UML metaclasses
$\mathcal{T}$	the names of all meta-attributes of the UML metamodel
$\mathcal{R}$	references to identifiers
$\mathcal{V}$	all base types defined by UML
$\mathcal{E}$	a set of tuples $t \mapsto v$ where t is a defined meta-attribute of $\mathbf{c}$ or any of its super meta-classes and v is the value of $\mathbf{c}$
$\mathcal{M}$	a set of tuples $i \mapsto e$ where i is the object identifier of an instance of a metaclass $\mathbf{c}$ defined in $\mathbf{m}$, and e is the element defined for each entity of m

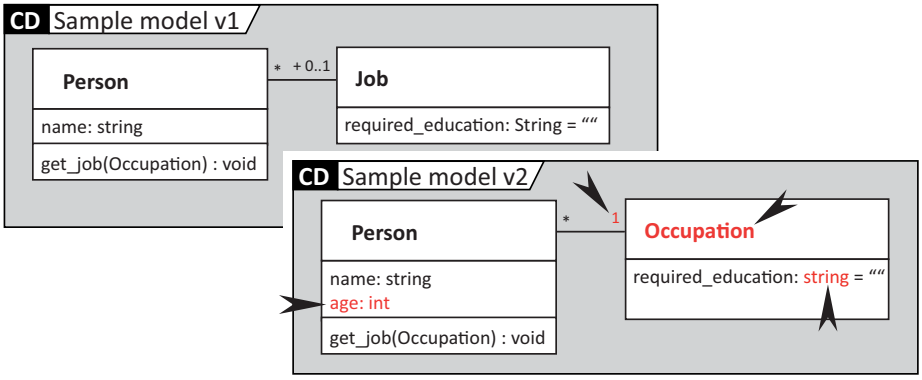
Fig. 2. Top: The instance $\mathbf{x}$ named "x" with identifier oid_x of the UML metaclass "Class" (left) is mapped into the mathematical structure on the right. Bottom: The domains used for mapping a metamodel based modeling language like UML.

For ARIS/EPC, the mapping is very similar, but it is difficult to make it as precise as the UML mapping since ARIS/EPC lacks a formal definition of the conceptual (and syntactical) framework: there is no such thing as a standard or metamodel. The easiest way to define a mapping for EPCs is probably the definition of a synthetic metamodel and then apply the rules defined above for UML analogously.

So, all eEPC concepts (such as event, function, connector, or role) become concrete metaclasses like in UML. Attributes of individual eEPC concepts are defined by the ARIS Toolset (or similar environments). The set of all possi-

ble attributes in any given setting is represented as the domain $\mathcal{T}$, the model population in any given model m is represented in the domain $\mathcal{I}$, and so on.

Now consider Figure 3 (top). Using the mapping sketched in the previous section, this simple UML class model may be transformed into a mathematical representation as shown in Figure 3 (bottom).



$$
\begin{aligned}
\{\quad & \#1 \mapsto \{\mathit{uml} \mapsto \text{class}, \mathit{name} \mapsto \text{Person}, \mathit{attributes} \mapsto [\#2, \#4], \mathit{operations} \mapsto [\#5]\},\\
& \#2 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{name} \mapsto \text{name}, \mathit{type} \mapsto \mathbf{string}\},\\
& \#4 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{multiplicity} \mapsto 0..1, \mathit{visibility} \mapsto \mathbf{public}, \mathit{type} \mapsto \#7\},\\
& \#5 \mapsto \{\mathit{uml} \mapsto \text{operation}, \mathit{name} \mapsto \text{get\_job}, \mathit{parameters} \mapsto [\#6], \mathit{result} \mapsto \mathbf{void}\},\\
& \#6 \mapsto \{\mathit{uml} \mapsto \text{parameter}, \mathit{type} \mapsto \#7\},\\
& \#7 \mapsto \{\mathit{uml} \mapsto \text{class}, \mathit{name} \mapsto \text{Job}, \mathit{attributes} \mapsto [\#8, \#9]\},\\
& \#8 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{name} \mapsto \text{required\_education}, \mathit{type} \mapsto \mathbf{String}, \mathit{default} \mapsto ""\},\\
& \#9 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{multiplicity} \mapsto *, \mathit{type} \mapsto \#1\},\\
& \#10 \mapsto \{\mathit{uml} \mapsto \text{association}, \mathit{ends} \mapsto [\#4, \#9]\} \qquad \}
\end{aligned}
$$

Fig. 3. Two versions of a sample model (top); formal representation of version 1 of the model.

3 Formalising operators and properties

In practical version control settings, a number of different problems occur, each requiring different operations. We will discuss three such settings in the remainder. For a function $f : X \to Y$, we call X the domain, denoted $dom(f)$. We consider functions as sets of pairs $\langle a, b\rangle$ where $a \in X$ and $b \in Y$, using the notation $x \mapsto y$ for its elements. Then, the usual set operations may be extended to binary functions, including set equality.

3.1 Difference

As the first scenario consider the computation of differences of models that have matching identifiers, that is, elements retain their identity under updates. We also assume that these identifiers are always globally unique. This is the case e.

g. for different subsequent versions of a UML model in XMI-based tools. With the previous definitions, we may define the following functions.

$$
\begin{aligned}
\mathit{added}(m_1, m_2) &= \{x \mapsto m_2(x) \mid x \in \mathit{dom}(m_2) \backslash \mathit{dom}(m1)\} \\
\mathit{removed}(m_1, m_2) &= \mathit{added}(m_2, m_1) \\
\mathit{changed}(m_1, m_2) &= \left\{ \begin{array}{l} x \mapsto m_2(x) \\ \quad x \in \mathit{dom}(m_1) \cap \mathit{dom}(m_2) \wedge m_1(x) \neq m_2(x) \end{array} \right\} \\
\mathit{common}(m_1, m_2) &= m_1 \cap m_2
\end{aligned}
$$

We define the predicate *conflicts* as an abbreviation for those model elements where the two models do not coincide.

$$
\begin{aligned}
\mathit{conflicts}(m_1, m_2) = \; & \mathit{added}(m_1, m_2) \\
& \cup \mathit{removed}(m_2, m_1) \\
& \cup \mathit{changed}(m_1, m_2)
\end{aligned}
$$

When the difference between two given models have been computed and any resulting conflicts have been resolved, the models must be merged. While it is quite natural to interpret (and treat) conflicts in $\mathit{added}(m_1, m_2)$ and $\mathit{removed}(m_1, m_2)$, there is more than one way to resolve conflicts collected in $\mathit{changed}(m_2, m_1)$.

Returning to the running example, compare versions 1 and 2 in Figure 3 of our sample class model. While nothing has been removed from v_1 to yield v_2 ($\mathit{removed}(v_1, v_2)$ yields the empty set), we find the following differences.

$$
\begin{aligned}
\mathit{added}(v_1, v_2) = \{ & \#3 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{name} \mapsto \text{age}, \mathit{type} \mapsto \textbf{int}\}\} \\
\mathit{changed}(v_1, v_2) = \{ & \#1 \mapsto \{\mathit{uml} \mapsto \text{class}, \mathit{name} \mapsto \text{Person}, \\
& \qquad \mathit{attributes} \mapsto [\#2, \#3, \#4], \mathit{operations} \mapsto [\#5]\}, \\
& \#4 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{multiplicity} \mapsto 0..1, \mathit{type} \mapsto \#7\}, \\
& \#7 \mapsto \{\mathit{uml} \mapsto \text{class}, \mathit{name} \mapsto \text{Job}, \mathit{attributes} \mapsto [\#8, \#9]\}, \\
& \#8 \mapsto \{\mathit{uml} \mapsto \text{feature}, \mathit{name} \mapsto \text{required\_education}, \\
& \qquad \mathit{type} \mapsto \textbf{String}, \mathit{default} \mapsto ''''\}\}
\end{aligned}
$$

These results inform us about the addition of the model element 3 in v_2, and changes to model elements 1, 4, 7, and 8. In order to examine more closely the changes made to these elements, the same functions may be used again since the contents of individual model elements are finite functions again. We simply select the contents of element x in models m and m' by $m(x)$ and $m'(x)$, and pass them to *changed* to yield the changed attributes and their values in m and m' as follows.

$$
\begin{aligned}
&\mathit{changed}(v_1(1), v_2(1)) = \{\mathit{attributes} \mapsto [\#2, \#4]\} \\
&\mathit{changed}(v_2(1), v_1(1)) = \{\mathit{attributes} \mapsto [\#2, \#3, \#4]\}
\end{aligned}
$$

Similarly, we analyse the changes for model elements 4 and 8. We find that in model v_2, attribute *visibility* of element 4 is removed, attribute *multiplicity*

of element 4 is changed from **0..1** to **1**, and the type of attribute 8 is changed from **String** to **string**.

$$\begin{aligned} \mathit{removed}(v_1(4), v_2(4)) &= \{\mathit{visibility} \mapsto \textbf{public}\} \\ \\ \mathit{changed}(v_1(4), v_2(4)) &= \{\mathit{multiplicity} \mapsto 0..1\} \\ \mathit{changed}(v_2(4), v_1(4)) &= \{\mathit{multiplicity} \mapsto 1\} \\ \\ \mathit{changed}(v_2(8), v_1(8)) &= \{\mathit{type} \mapsto \textbf{string}\} \\ \mathit{changed}(v_1(8), v_2(8)) &= \{\mathit{type} \mapsto \textbf{String}\} \end{aligned}$$

The difference for model element 7 is structurally similar to that of element 8.

3.2 Conflict resolution and merging

Let us assume for a moment, that the conflict resolution has left us with two models m_1 and m_2 without conflicts, then merging two models is simply set union, that is, the simplest merge function merge_0 is defined as follows.

$$\mathit{merge}_0(m_1, m_2) = m_1 \cup m_2$$

On the other hand, if there are conflicts between m_1 and m_2, these need to be resolved before m_1 and m_2 may be merged. Possibly the simplest conflict resolution and merge strategy ("merge_1") is to give priority to one of the two models.

$$\begin{aligned} \mathit{merge}_1(m_1, m_2) = \; & \mathit{common}(m_1, m_2) \\ & \cup \; \mathit{added}(m_1, m_2) \; \cup \; \mathit{removed}(m_1, m_2) \\ & \cup \; \{i \mapsto m_1(i) \mid i \in \mathit{dom}(\mathit{changed}(m_1, m_2))\} \end{aligned}$$

Here, common elements and elements unique to either m_1 or m_2 are collected. When there are different values for some identifier, m_1 takes priority over m_2.

A more elaborate strategy applies this schema recursively and merges individual attributes of model elements, too, yielding the following definition.

$$\begin{aligned} \mathit{merge}_2(m_1, m_2) = \; & \mathit{common}(m_1, m_2) \\ & \cup \; \mathit{added}(m_1, m_2) \; \cup \; \mathit{removed}(m_1, m_2) \\ & \cup \; \{i \mapsto \mathit{merge}_1(m_1(i), m_2(i) \mid i \in \mathit{dom}(\mathit{changed}(m_1, m_2))\} \end{aligned}$$

Recall that, formally, individual model elements are (finite) functions just like models. Using merge_2 recursively inside the right hand side of the definition would yield an infinite recursion. Using merge_1 instead avoids this problem.

Observe that all of the merge strategies mentioned so far are identical, if there are no unilateral changes to shared elements, that is,

$$\mathit{merge}_0(m_1, m_2) = \mathit{merge}_1(m_1, m_2) = \mathit{merge}_2(m_1, m_2)$$

if $changed(m_1, m_2) = \emptyset$. In other words, if concurrent work on shared model elements can be avoided (e. g. by locking or organisational measures), the problematic case does not occur at all.

More complex strategies may be implemented, too. However, very specific heuristics are likely to be applicable to only one modeling language or method. The alternative, then, are semi-automatic strategies involving manual work. Whether there are more powerful yet general strategies will have to be determined empirically. In the remainder, we will use *merge* to represent an arbitrary merge-function.

3.3 Matching and similarity/distance

In the scenarios above, we have assumed the presence of globally unique and appropriately matching identifiers of model elements which were not affected by updates to the model element. This is, obviously, not always the case. For instance, some tools do not support identifiers other than the names of model elements (including well-established commercial tools such as Matlab/Simulik™ and BOC Adonis™) which of course are subject to changes. In other cases, elements might get lost by copy/paste-operations. In yet other cases, the models that are to be compared may not be different branches of a common base model.

In such cases, the assumption of matching identifiers does not hold. Thus, in order to be able to apply the previously defined operations, a mapping between the identifiers of the two models must be computed first. In other words, we need to find an "optimal" bijection between the identifiers of two models, use this to consistently relabel (the identifiers of) one of the models, and proceed as before.

The term "optimal" implies the existence of a better-than-relationship on mappings. Therefore, we define a measure $binding : \mathcal{M} \times \mathcal{M} \rightarrow I\!R^+$ that quantifies the "binding strength" between two models as their overlap. For this, the function *binding* is defined for all the domains we have introduced above.

$$binding_{\mathcal{B}}(v, v') = \begin{cases} 1 & \text{if } v = v' \\ 0 & \text{otherwise} \end{cases}$$

$$binding_{\mathcal{R}}(i, i') = \begin{cases} 1 & \text{if } i = i' \\ 0 & \text{otherwise} \end{cases}$$

$$binding_{\mathcal{S}}(s, s') = \frac{|s \cap s'|}{|s \cup s'|}$$

$$binding_{\mathcal{E}}(e, e') = \sum_{tag \in dom(e) \cap dom(e')} [e(tag) = e'(tag)]$$

$$binding_{\mathcal{M}}(m, m') = \sum_{id \in dom(m) \cap dom(m')} binding_{\mathcal{E}}(m(id), m'(id))$$

where the notation $[a = b]$ is used to denote 1, if $a = b$ and 0 otherwise (cf. [9, p. 24]).

Using this simplistic binding metric, matching is an optimization problem: given two models m and m, find a relabeling r such that $binding(r(m), m')$ is maximal. More complex similarity algorithms exist in the field of bioinformatics (e. g. FASTA [19]).

The binding relative to the number of binding sites (that is, tags), is used to define similarity between model elements and models.

$$similarity_{\mathcal{E}}(e, e') = 2 * \frac{binding_{\mathcal{E}}(e, e')}{|e| + |e'|}$$
$$similarity_{\mathcal{M}}(m, m') = 2 * \frac{binding_{\mathcal{M}}(m, m')}{|m| + |m'|}$$

As one would expect, for both these definitions we have $similarity(x, \emptyset) = 0$ and $similarity(x, x) = 1$.

3.4 Versioning phenomena

Based on the previous definitions we may now describe precisely some phenomena occurring in version control.

clones Two model elements e_1 and e_2 are clones if they have different identifiers but are very similar or identical with respect to their contents ($similarity_{\mathcal{E}}(e_1, e_2) \geq 0.9$).

relatives Two model elements e_1 and e_2 are relatives if they have different identifiers but are rather similar ($0.5 \leq similarity_{\mathcal{E}}(e_1, e_2) \leq 0.9$).

mates Two model elements e_1 and e_2 are mates if they have different identifiers and are not very similar ($0.1 \leq similarity_{\mathcal{E}}(e_1, e_2) \leq 0.5$).

foreigners Two model elements e_1 and e_2 are foreigners if they have different identifiers and are not similar ($0.1 \leq similarity_{\mathcal{E}}(e_1, e_2) \leq 0.5$).

identity Two model elements e_1 and e_2 may be considered identical only when they have the same identifier, and identical or almost identical content ($similarity_{\mathcal{E}}(e_1, e_2) \geq 0.9$). Using a threshold smaller than 1 does not interfere with detecting and merging changes. Similarity beyond this threshold (but below 1) is typically only achieved by minuscule changes (such as an updated timestamp or status flag).

mutation Two model elements e_1 and e_2 with the same identifier, and similar but not identical content ($0.5 \leq similarity_{\mathcal{E}}(e_1, e_2) \leq 0.9$), are likely to be mutations from a common predecessor, or one a mutation of the other.

evolution Two model elements e_1 and e_2 with the same identifier, that also share a substantial but not massive amount of other features ($0.1 \leq similarity_{\mathcal{E}}(e_1, e_2) \leq 0.5$), are likely to have evolved from a common predecessor, or one has evolved from the other.

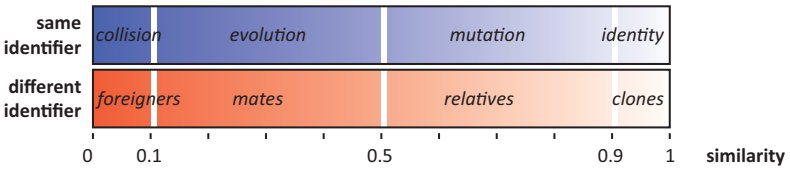


Fig. 4. Synopsis of version conflict phenomena

collision Two model elements e_1 and e_2 with the same identifier but little or no similarity otherwise ($0 \leq similarity_{\tau}(e_1, e_2) \leqslant 0.1$) collide.

These phenomena are presented synoptically in Figure 4. When two models are to be merged, these categories of phenomena may be used to classify all resulting merge conflicts. Unfortunately, for all these phenomena there are scenarios that require different measures. Therefore, faultless automatic resolution is not possible. However, classifying the collisions between two models, and visualising and presenting them accordingly enormously facilitates manual conflict resolution and merging of models.

4 Implementation

We have created the Model Manipulation Toolkit (MoMaT) to implemented our approach based on a incomplete UML-2-profile for the BOC ADONIS-tool. It exports a proprietary comma-separated-value format. The UML-2-profile is structurally very unlike the original UML metamodel, but does share names with UML. We have also implemented mappings from various combinations of UML and XMI-versions for the Tools NoMagic *MagicDraw UML*, *Fujaba*, and Sparx Systems' *Enterprise Architect*.

In our approach, models from all sources are transformed into the common meta-metamodel for differencing which is then stored as Datalog clauses in a Prolog database. In a second step, the differences between models are computed directly on this database (see Figure 5).

Technically, instances of this Meta-metamodel are encoded as sets of Datalog-clauses (cf. [7]), that is, as simple logical predicates that could also be mapped into a traditional relational schema. In this transformation, each model element becomes a Prolog-fact of the form

```
me(type-id, [tag-value, ...]).
```

where `type` is the UML metaclass, `id` is an arbitrary unique identifier (such as an integer or term of integers), `tag` is any atom to identify an attribute of the model element, and `value` is the value of this attribute. References are marked by the terms `ref` and `refs`, respectively. Thus, version 2 of the UML class diagram of Figure 3 transforms into the following Datalog clauses.

```
me(class-1,       [name-'Person', attributes-refs([2, 4])]).
me(feature-2,     [name-name, type-string]).
```

```
me(feature-4,      [multiplicity-'0..1', type-ref(7)]).
me(operation-5,    [name-'get_job', parameters-refs([6]), result-void]).
me(parameter-6,    [type-ref(7)]).
me(class-7,        [name-'Job', attributes-refs([8, 9])]).
me(feature-8,      [name-'required_education', type-string, default-'']).
me(feature-9,      [multiplicity-*, type-ref(1)]).
me(association-10,[ends-refs([4, 9])]).
```

The value of attributes may be arbitrary Prolog-terms. When the values of all tags in a model element are either atoms, lists of atoms, or terms `set(x)` where `x` is one of the former, the model element is said to be in normal form. When all model elements of a model are in normal form, the whole model is in normal form.

It has been easy to define predicates for computing the similarity of model elements, the difference of models, and so on. Due to the declarative nature of prolog programs and and the very simple data structure, there is a direct and traceable correspondence between the mathematical definitions and the implementation allowing for easier checking and understanding. The correspondence between the domains defined above and the implementation is obvious which is particularly helpful when experimenting with different mathematical concepts to formalize properties and operations on models.

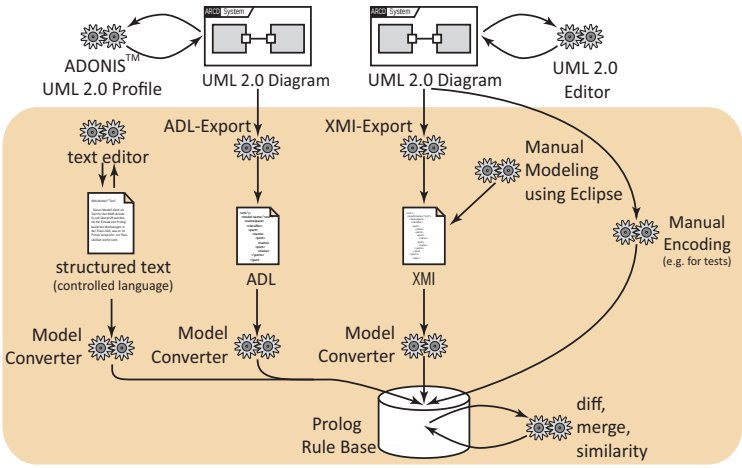


Fig. 5. The architecture of MoMaT: different converters may be used to upload models from a variety of tools into a common model repository. Version-management operations are then defined on this repository.

5 Experiences and discussion

We have applied our approach and implementation experimentally in two very large scale industrial projects with the German federal Tax Authority and the German Public Pension Authority. The first experiences have been very encouraging; to a degree, in fact, that our industrial partners have licensed our technology and are actively seeking to increase the area of application of our approach.

However, there are several action items that need further research. For instance, we do need more empirical data in order to calibrate the thresholds used in section 3.4. Also, we still rely on the existence of globally unique identifiers. Computing matches if these are missing is a very time consuming procedure in our current implementation, and error rates are currently not satisfactory.

The main feature of our approach is its generality: it is applicable to all modeling languages and data formats, as long as a mapping to the meta-meta-model of Figure 1 can be established, a very light restriction indeed. However, this generality and versatility comes at a price, as [24] has already remarked.

Another area that asks for improvement is the visualisation of differences. We are currently investigating to apply and modify the scheme presented in [16].

Acknowledgments We would like to express our gratitude to the reviewers for their helpful comments.

References

1. Dirk Ahrens. Differenzanalyse und Vereinigung von Modellen auf der Basis ihrer Metamodelle. In Kelter [12]. appeared in Softwaretechnik-Trends 2(27)2007.
2. Marcus Alanen and Ivan Porres. Difference and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc.* 6th *Intl. Conf. Unified Modeling Language (<<UML>>'03)*, volume 2863 of *LNCS*, pages 2–17. Springer Verlag, 2003.
3. Philip A. Bernstein, Alon Y. Levy, and Rachel A. Pottinger. A Vision for Management of Complex Models. Technical Report MSR-TR-2000-53, Microsoft Research, June 2000. also appeared as SIGMOD Record 29, 4 (Dec. '00).
4. Ray J. A. Buhr. Use Case Maps as Architectural Entities for Complex Systems. *IEEE Transactions on Software Engineering*, 24(12):1131–1155, December 1998.
5. Mario Bunge. *Treatise on basic philosophy: Ontology I.* Reidel, 1977.
6. Mario Bunge. *Treatise on basic philosophy: Ontology II.* Reidel, 1979.
7. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases.* Surveys in Computer Science. Springer Verlag, 1990.
8. Rob Davis. *Business Process Modelling with ARIS: A Practical Guide.* Springer Verlag, 2001.
9. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1991. seventh printing, with corrections.
10. Volker Gruhn and Ralf Laue. Validierung syntaktischer und anderer EPK-Eigenschaften mit PROLOG. In Markus Nüttgens, Frank J. Rump, and Jan

Mendling, editors, *Proc. 5. GI Ws. Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten (EPK 2006)*, volume 224 of *CEUR Workshop Proceedings*, pages 69–85, Bonn, December 2006. Gesellschaft für Informatik.
11. Christian Hein and Tom Ritter. Versionierung für Modellgestützte Entwicklung über Werkzeuggrenzen hinweg. In Kelter [12]. appeared in Softwaretechnik-Trends 2(27)2007.
12. Udo Kelter, editor. *Proc. Ws. Versionierung und Vergleich von UML Modellen (VVUU'07)*. Gesellschaft für Informatik, May 2007. appeared in Softwaretechnik-Trends 2(27)2007.
13. Sergey Melnik, Philip A. Bernstein, Alon Y. Levy, and Erhard Rahm. A Semantics for Model Management Operators. Technical Report MSR-TR-2004-59, Microsoft Research, June 2004.
14. Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *Proc. ACM Intl. Conf. Management of Data (SIGMOD'03)*, pages 193–204, June 2003. available at `www.websemanticsjournal.org/volume1/issue1/Melniketal2003/index.html`.
15. Michael Meyer and Helge Schulz. SHORE: Ein Werkzeug für übergreifende Vernetzung und Auswertung von Dokumenten. In Franz Ebert, Jürgen und Lehner, editor, *Proc. Ws. Software-Reengineering*. Gesellschaft für Informatik e.V. May 1999.
16. Jörg Niere. Visualizing differences of UML diagrams with Fujaba (Position Paper). In *Proc. Intl. Fujaba Days 2004*, September 2004.
17. Daniel Ohst, M. Welle, and Udo Kelter. Differences between versions of UML diagrams. In *Proc. 3rd Eur. Software Engineering Conf. 2003 (ESEC'03)*, pages 227–236, September 2003.
18. OMG. UML 2.1.1 Superstructure Specification (formal/ 2007-02-03). Technical report, Object Management Group, February 2007. available at `www.omg.org`, downloaded at May 25$^{\text{th}}$, 2007.
19. T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.
20. Martin Soto. Delta-P: Model comparison using semantic web standards. In Kelter [12]. appeared in Softwaretechnik-Trends 2(27)2007.
21. Harald Störrle. *UML 2 erfolgreich einsetzen*. Addison-Wesley, 2005.
22. Harald Störrle. A approach to cross-language model versioning. In Kelter [12]. appeared in Softwaretechnik-Trends 2(27)2007.
23. Harald Störrle. A Logic-based Approach to Representing and Querying Software Engineering Models. In Phil Cox, Andrew Fish, and John Howse, editors, *Proc. Intl. Ws. Visual Languages and Logic (VLL'07)*, 2007. to appear.
24. Bernhard Westfechtel. Differenceing and merging of Software Diagrams. In Kelter [12]. appeared in Softwaretechnik-Trends 2(27)2007.
25. Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *Proc. Intl. Conf. Automated Software Engineering (ASE'05)*, pages 54–65. ACM Press, 2005.

(H)ALL: a DSVL for designing user interfaces for Control Systems.

Bruno Barroca
Universidade Nova de Lisboa
bruno.barroca@di.fct.unl.pt
and Vasco Amaral
Universidade Nova da Lisboa
vasco.amaral@di.fct.unl.pt

Abstract. This paper presents the methodology used to design a Domain Specific Visual Language (DSVL), named HALL (Human Assisted Logic Language), for the purpose of deriving automatically or semi-automatically user interfaces for critical control systems. We will describe our approach to design this language's abstract syntax and the process to describe its semantics by using formal framework based on an algebraic Petri-nets formalism: Concurrent Object-Oriented Petri Nets (CO-OPN).
Keywords: Coordination-engine, Queries-Views-Transformations (QVT), interface model components, petri-net, CO-OPN, control systems, finite state machine, message model.

1 Introduction

Control systems are hierarchical systems composed by several subsystems and components. On the leafs of the hierarchy, typically are the sensors and controls which may deal with continuous and discrete values at various refresh rates in a synchronous and asynchronous fashion. Control Systems used to be centralised in a single operational control centre (OCC), nowadays they tend to be distributed. The control interfaces can be deployed on several machines on each level of a control system. Lower level interfaces are usually more specialised and perform control and monitoring with a great level of resolution and accuracy. Top level interfaces are usually less specialised, but presents statistical information providing a comprehensive global view of the overall performance of the system.

Typically, the development of user interfaces for critical control systems consists of a complex and slow process. The problem tends to scale with the increasing complexity of the systems involved. For instance in the high energy physics domain, a detector machine is composed by thousands of different components which have to be controlled and exposed to the operators in a intuitive and usable way. To develop individually the graphical user interfaces (GUIs) is not viable in time.

In order to tackle this problem, the GUI development process should be more systematic by allowing the domain expert to start at the requirements and

interface specification levels, and afterwards deploy the artifacts in a automatic or semi-automatic way. Together with this approach, several extra benefits will come, such as verification (model correction), and validation (assuring that the model covers completely the initial requirements).

The BATIC3S[1, 8, 9] project aims to build a comprehensive methodology for semi-automated, formal model-based generation of effective, reliable and adaptive 3D GUIs for diagnosing control systems. Within this context, there was the need to define a domain specific visual language (DSVL) with which we could support interface modeling and integrate with the framework, based on a formal semantics ground by using Petri-Nets, for fast prototype generation.
The (H)ALL is an approach to that problem. It uses the Finite State Machine (FSM) formalism to express behaviour of both system and GUIs. Current FSM models are limited since they lack a hierarchical structure and they do not possess any formal mechanism of synchronisation between several FSM's. To effectively express some problem in a specific domain, one usually decompose the problem (divide) into several logical components which may seem independent in one of more aspects (dimensions). Each component can again be decomposed into several levels of abstraction. Finally all the components and subcomponents need to be synchronised and updated to compose a behaviour model required to express the solution to the initial problem (conquer). Hierarchical FSM's achieve FSM composition. In (H)ALL, FSM's are synchronised using a message broadcasting mechanism. Furthermore, (H)ALL is subdivided into two components.
The first one is exclusively dedicated to model the structure and behaviour of the system itself. It can be used to simulate and verify the lower level interfaces with the real system (drivers). Although it can also be used to aid the very definition of a user interface model, and simulate the generated GUI artifacts without requiring to plug-in to the real control system. The connection with the real system is not really tackled on this work. Here we assume that it will be assured by platform-dependent drivers which will interface with our coordination engine.
The second one represents the users and their profiles. For each profile, we will model the view that each user will have over the control system. Here it is possible to define their specific GUIs, specific tasks and specific user permissions and roles on each level of the control system.

This paper focuses on the approach and methodology to derive a DSVL named HALL (Human Assisted Logic Language), as well as on its formal definition. We have chosen the drink vending machine (DVM) [4] as a case study to verify and validate the current work. Some of samples presented along this paper are based on the DVM case study analysis.

In the rest of the paper, we will have a look at related work in section 2, then we will focus on the language design issues in section 3 and finally we will discuss future developments of our ongoing work in section 4 .

2 Related Work

The research area of user interface prototyping is vast and broad. However, to our knowledge, there is a lack of bibliographic references presenting a comprehensive survey in this specific area of control systems applications.

As shown in Table 1, in what concerns to GUI prototyping, there is a plethora of work done: TRIDENT[10], MOBI-D[6], ERGOCONCEPTOR[5], ENVIR-3D[11], to name a few. The underneath concept is a powerful mechanism to speed up the design process by automatically generating a user interface based on a formal specification of some kind. We will next summarise the mentioned projects under the dimensions we found relevant for this work: prototype generation and modeling capabilities.

Project	Domain	Application
TRIDENT	2D GUIs	General scope
MOBI-D	2D GUIs	Business applications
ERGOCONCEPTOR	2D GUIs	Process control applications
Envir3D	3D GUIs	Control systems

Table 1. Researched projects on the domain of GUI prototyping.

Prototype generation - Prototype generation is achieved by means of mechanisms built to ease and fasten the production of GUIs based on some sort of specification (it may not be a formal one). TRIDENT presents full GUI prototype generation in several aspects: it uses configurable production rules to guide the designer, and automatically take some options on the interface design. The positioning of the widgets on the dialogues is automatically done given the desired layout strategy, effectively reducing the design time of a user interface. Clearly, it is shown that there must be a balance in a Rapid User Interface Prototyping (RUIP) system between flexibility and the design time spent on specifying all the details of the user interface's design. MOBI-D uses a tool (TIMM) to assist the designers in their design choices, not limiting however, the artistic options that the designer may take. ERGOCONCEPTOR balances the capability of rapidly generate a first version of the GUI formed by abstract user controls (and with several design options left open) to the flexibility of instantiating those abstract user controls in an external design editor - this way joining the best of two worlds. Finally, Envir3D can also generate a user interface (VRML code) based on a XIML specification. Their Abstract Interface Objects (AIO) library can effectively speed up the design process in a considerable way.

Modeling - A GUI design framework is said to be model-based if it implements some of these interface model components: presentation model (dialogue and geometric model are usually included here), task model, user profile model, domain specific model. Some also include a design model which resembles interface model components composition issues MOBI-D seems to be the most

general-purpose of all the studied model-based systems, as it is the one which implements all the components of the *interface model*. Both TRIDENT (business oriented) and ERGOCONCEPTOR (process control oriented) imposes a specific design order which also guides the designer through all the design process. MOBI-D also drives the design process from abstract models to concrete ones, but it uses a specific model (design model) to glue link all models and an assistant (TIMM) to help the designer pruning the space of choices. This way of organising the mappings can be interesting if we are to support iterative design due to the fact that all modeling components are intrinsically independent from each other. Finally in Envir3D, they have limited their work to presentation and dialogue models, but they left doors open to the other interface model components by adopting the XIML format on their model-based engine. Envir3D presents a unified model to coordinate all aspects of the interface model, however they do not give a clear answer from how do they deal with the aspects of concurrency and parallelism, nor give an architectural view of all involved modules of their prototyping solution.

Final Remarks - From all the work that we have researched on the fields of GUI prototyping and GUI modeling, only Envir3D has presented an unified model for GUI specification specifically in the domain of control systems. However, we require a methodology based on a complete and strong formal basis, to deal will all aspects of prototyping a GUI in a uniform and intuitive way.

3 (H)ALL Design

The purpose of designing a new domain specific visual language (DSVL) is to enable designers and system engineers to think and develop using the very own terms and sentences of the problem's domain instead of solution's domain. On the other hand, general purpose languages (GPL) available nowadays, use terms and sentences that usually concerns to software engineering concepts like reuse, modularity, etc.

The (H)ALL design framework will attempt to give to designers an integrated design view, centring on each object/component. In each design step, the language below automatically separates aspects and concerns like user profiles, graphical objects and tasks.

We ought to create both kinds of 2D and 3D interfaces to control systems. These interfaces are composed by commonly known widgets like combo-boxes, radio-buttons, push buttons, context menus, and others with drag and drop capabilities, auto-resizing, auto-placing and other interaction behaviour.

In the Figure 1, it is shown an example of a generated GUI running on a graphics engine which was developed specifically in the context of BATIC3S project.

The (H)ALL is intended to feed this particular engine with data containing the geometrical information about the interface objects and their visual/interaction properties. The particular behaviour of each widget can be either fully modeled in (H)ALL, or just left has a property/parameter for the designer to configure.

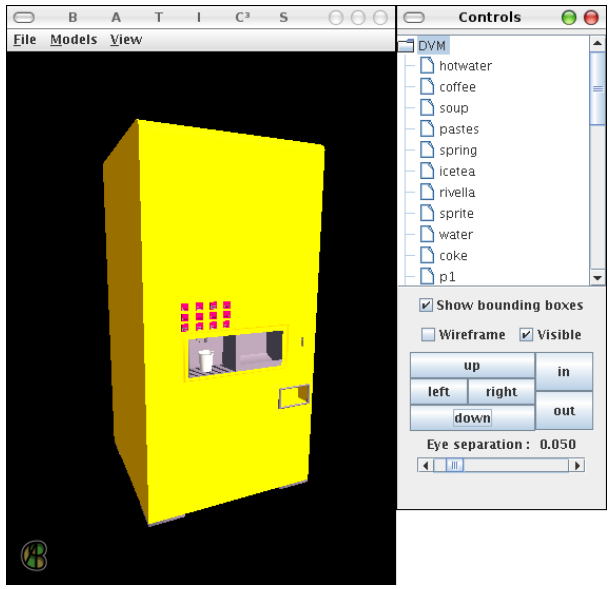


Fig. 1. Prototype of GUI for a Control System.

The (H)ALL visual model only reaches the notion of the building of a logical GUI. The specified widgets (it's structure and behaviour - states and events) are abstract, and can be instantiated in any graphics engine you choose to implement. Therefore, the concrete interfaces generated by (H)ALL are strongly dependent on the final implementation of the graphics engine in a particular platform.

3.1 Language Syntax Specification

The (H)ALL syntax specification is divided into a "hidden" textual formalism and a visual formalism, which are both closely connected on the design process. In this paper, due to space concerns, we only describe the (H)ALL's abstract syntax, excluding a description of the concrete visual syntax that is described in [2].

The (H)ALL abstract syntax is divided into two parts: the kernel structures and the domain structures, which can be considered as sub-languages. Domain structures attempt to model the different aspects of building a GUI (see Figure 2).

When specifying a GUI for a control system the designer has to take into consideration several aspects like: who will use the GUI (defining it in the users model), the GUI requirements (defining it in the task model), the GUI structure (dialogues, widgets, etc..) and behaviour (dynamism and properties, defined in the visual model). Last and not least, we include a system model which enables the designer to model a simulator of the real control system for simulation,

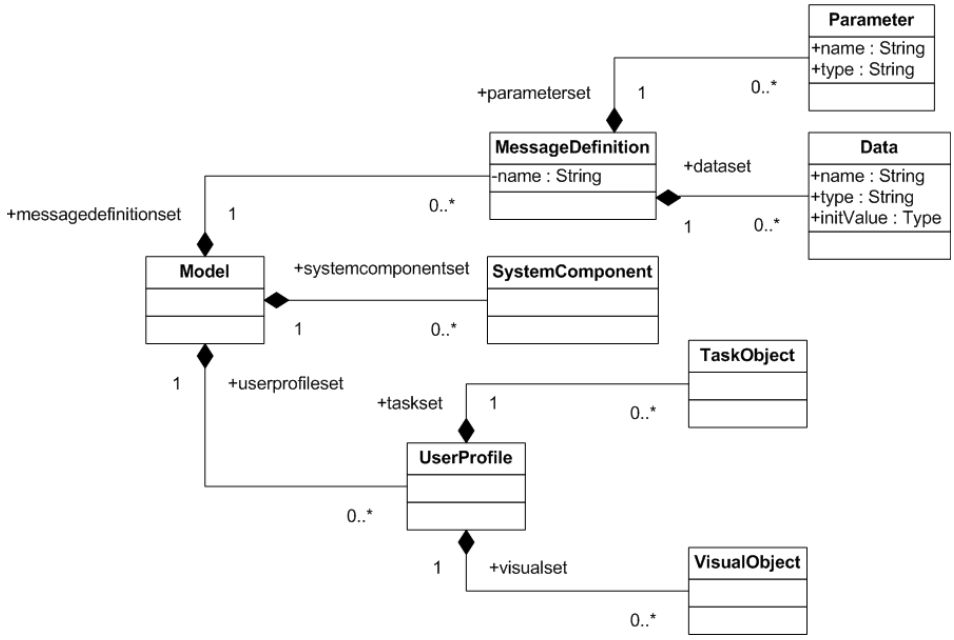


Fig. 2. (H)ALL's top level meta-model.

testing and verification purposes. All these models are designed using the state machine abstraction. State machines are synchronised using a messaging model.

The hierarchical state machines and messaging model forms what we name the Kernel. Sub-domain specific behaviour is defined on external machines which are connected with the Kernel by the respective domain structures. For instance, the GUI objects geometrical specifications defined in the visual model will feed the graphics engine (built on some platform) which implements the graphical object rendering. In Figure 3 it is shown the (H)ALL structures used to specify GUI objects.

The same for user profiling specific data that will feed the User Interface database management engine (IUD). As shown in Figure 4, (H)ALL is used to define the structure and behaviour of the coordination between all the components which compose the interface model. It achieves that by both modeling and simulating the internal states of each engine, and by synchronising all engines using messages.

The Kernel The (H)ALL, as many other languages, has a core level/layer, which is named the Kernel. It resembles most of the common properties on the design process of each sub-language.

In Fig. 5 it is shown the component's meta-model, which is here referred as the Kernel. The Kernel can be divided into two parts. The first one is the Finite State Machine (FSM), and the second is the messaging. In each hierarchy of

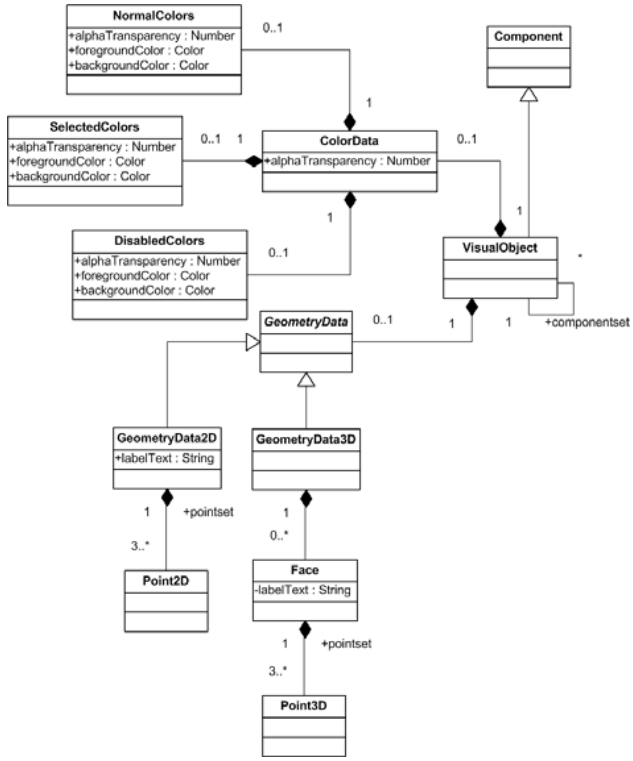


Fig. 3. (H)ALL's visual meta-model.

components among all sub-domains, each one (each node) will have a model of its behaviour described as a state machine. The state machine is a graph-like model specification of the behaviour of common systems. It is composed by states and transitions, which will be given values in each sub-domain specification. A transition is a relation between states. It can be defined as a tuple of two states (source state, target state). Here we simplify to simple graphs, but we can always power-up (by composition) to an equivalent semantic of hyperbolic state graphs.

On the FSM model editor, states are connected each other by named transitions. Each transition includes a trigger, a precondition, a postcondition and an action. The trigger activates the transition. Triggers can be either events from the domain, or message notifications. After the transition is activated by a trigger, the transition's precondition is evaluated as a boolean expression. If the precondition evaluates to "true", the transition is fired and the active state is changed from the origin state to the target state. After this, the postcondition is evaluated (setting internal values). After postconditions are evaluated, actions are then executed. In actions, it is possible, for instance, to set domain properties or send messages to other components. These messages don't have a particular destination: a message is broadcast along the hierarchy, and can be handled by any component which implements a handler for it.

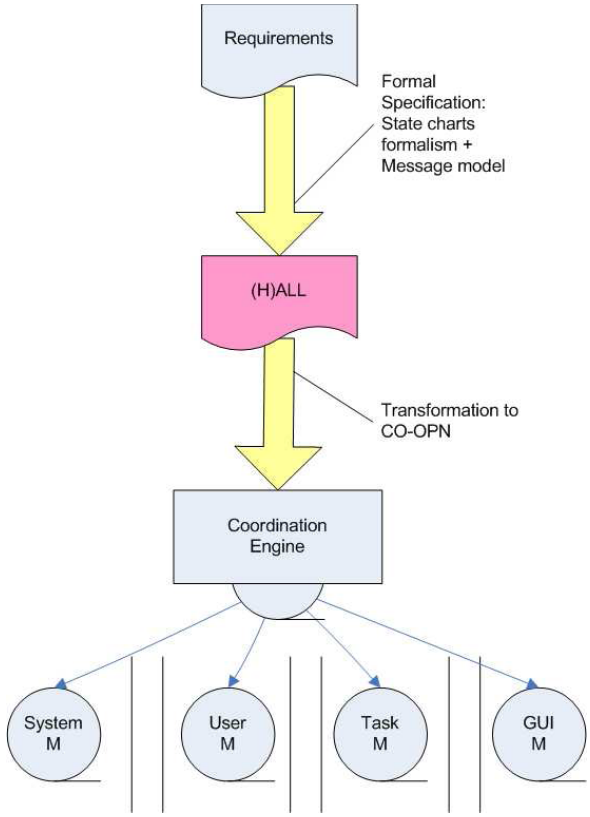


Fig. 4. GUI development process using (H)ALL.

Messaging - The messaging model is one of the most important facets of the design process, since it implements the communication between all sub-domains.

A message is an entity which is able to be cloned over the system's structure. A message has it's internal data structures and it's own FSM. The message FSM will only run at the time the message is being handled. This FSM is therefore contextual to the message being handled, and will provide direct access to manipulate not only component data, but message parameters and data as well. However, there are predefined states which enable the designer to control the message routing over the domain hierarchy. The propagation mechanism should be (on designers view) an implementation detail. For now let's consider it will run like the virtual table mechanism on C++. By default, a message is passed to children or to parents till some component handles it.

The messages are passed between engines, thus it is natural that the message definitions have to be placed on a global area, to be accessed by all sub-domains. In the message definition area is where the designer defines messages signatures and its internal data structures and types. It is also on this area where the designer specifies the message data initialisation behaviour. A signature is the

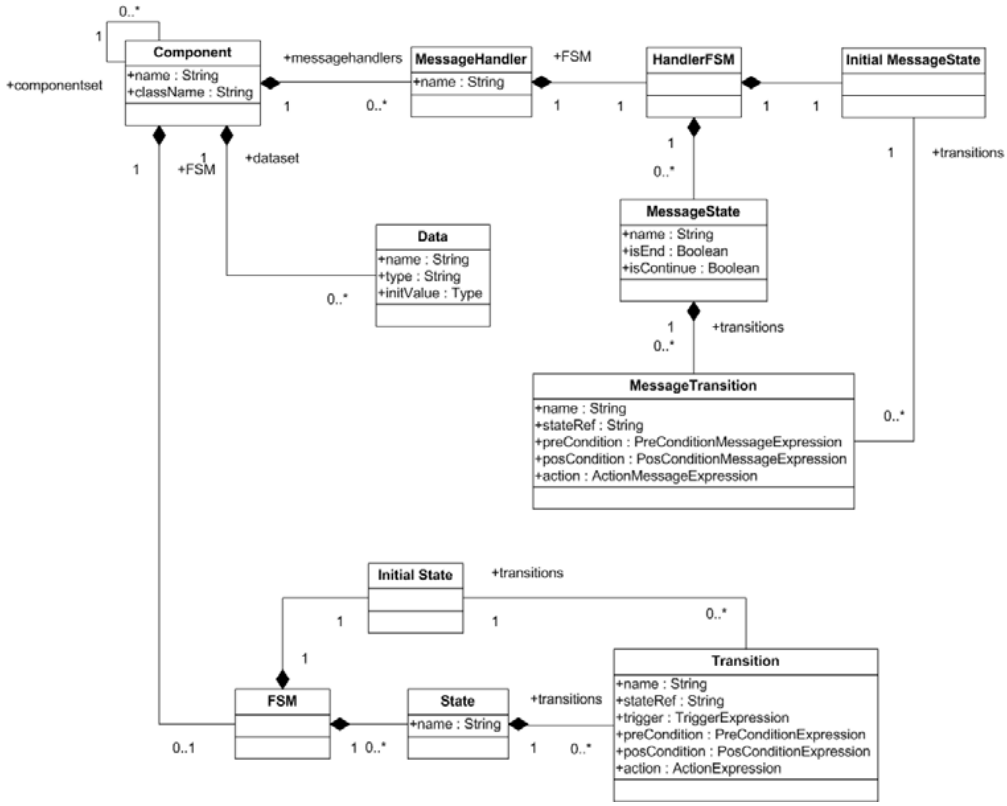


Fig. 5. Component Model in (H)ALL

name of the message and its parameters. It is then used by message handlers which implement the receiving behaviour of that message of that specific signature. Messages from one engine can be fired an handled by other engine, simply by using the **MessageInvocation** expression. Message handlers of a certain component can be all disabled using the **Enable** expression or one can specify exactly what handler is to be enabled/disabled.

On a component hierarchy, each node will have defined several message handlers. These message handlers have themselves internal state machines which express the behaviour of receiving a message. Typically, while specifying a message handler, the designer specifies the processing of the message data, merging that information with the internal components data or its FSM, and will then decide about if the message was well received. If so, it can be signalled to the component inner FSM (unblocking all *MessageNotification* and consequently triggering state transitions). When a message is sent, the message state is automatically set to "continue". The state of a message is automatically set to "stop" whenever it reaches a component which implements an handler to that message. Otherwise, the message stays in the "continue" state. If no component

implements an handler for it, the message will be silently lost. At this point the designer is also able to reset the routing direction "children to parents" or "parents to children", by setting the message state with the property "topdown".

Since the message handler's FSM is activated whenever the message arrives to the component, there is no trigger to be defined on the message handler's FSM transitions. Also, a message handler cannot expect message notifications from another message, because it is not possible (by design) to send a message to a message handler. Note that in a message handler it is possible to access to message's data and parameters, as well as component's data.

3.2 Language semantics specification

The (H)ALL semantics has just been informally described. Here we will describe the methodology used to formally specify the its semantics.

The specification of the semantics was defined [2] by model transformation to a known semantics language: CO-OPN [3](which stands for Coordinated Object-Oriented Petri-Nets). The CO-OPN language is already a graph-like language, so one could think to use it instead. However, CO-OPN is a General Purpose Language (GPL), so writing a user interface would be a lot more complicated and error prone. Here we express (H)ALL in terms of CO-OPN constructs which could rather be directly implemented in CO-OPN.

The translation focuses on the Kernel constructs and was made only for semantic specification purposes, which means that it is not an optimised translation, and the resulting CO-OPN code is not efficient. Note that performance requirements are an issue while designing user interfaces to critical and complex control systems. Thus, performance is an issue that we must deal afterwards. Here we just want to formally specify the semantics of (H)ALL.

To demonstrate the transformation methodology, we will show how an (H)ALL system component - DVM - is transformed into a CO-OPN context (see Figure 6). Note that, for simplicity, the proposed visual syntax do not show all contents in detail [2]. The system component DVM is composed by two subcomponents: Money Unit and Container. Also, DVM has its own FSM (including three states) and it implements a message handler FSM to message Shutdown.

The transformation specification expressed in QVT (Query-View Transformation)[7] rules uses the meta-model definition of the (H)ALL as source model and the meta-model definition of the CO-OPN language as target model. Therefore, the resulting model will be a CO-OPN specification of a GUI.

Top-level The transformation rules in QVT (Query-View Transformation) starts to flat (H)ALL components hierarchy into CO-OPN contexts. Thus, component compositions is mapped into CO-OPN *context-use* compositions. All top-level components in a (H)ALL model are mapped into CO-OPN contexts and *context-used* in a default CO-OPN context named "**SpaceModelCTX**". For every defined message in (H)ALL, all CO-OPN context translated from (H)ALL components will have a method (*MessageInPort*) and a gate (*MessageOutPort*) to

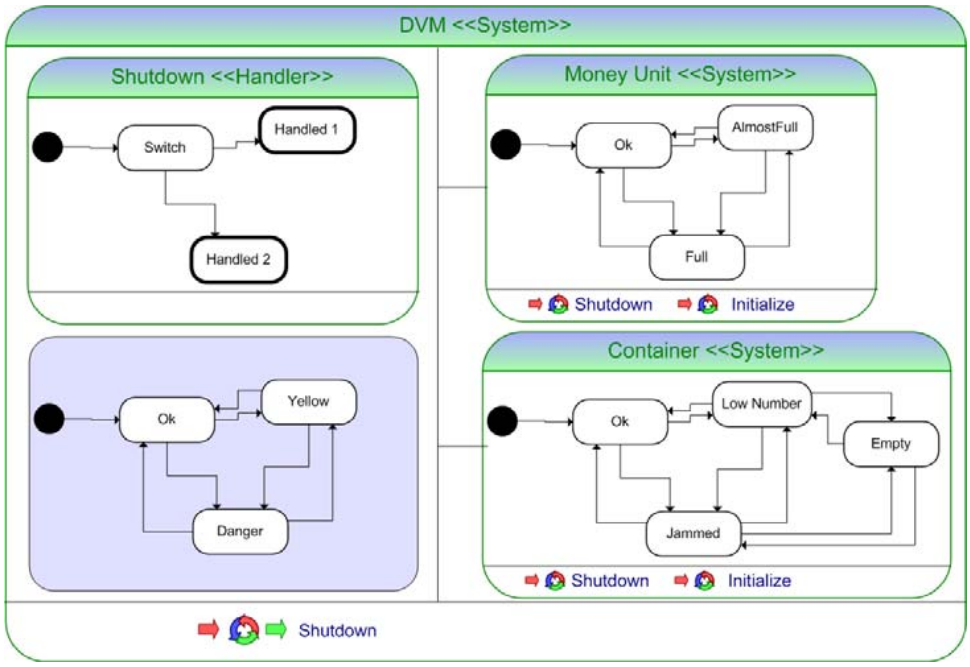


Fig. 6. DVM system component expressed in (H)ALL

transport that message along in the CO-OPN context hierarchy. Note at this point, that this translation is not optimised since there could exist components that would never need to route messages of that kind.
Then, in the **SpaceModelCTX** context, all defined messages in (H)ALL are translated into CO-OPN synchronisations between gates (*MessageOutPort*) and methods (*MessageInPort*) of every defined top-level contexts, forming a completely connected graph. This synchronisation allows the passing of messages between domains (e.g. passing a message from system to GUI, or vice-versa).

Inside Component The resulting CO-OPN model of several transformations applied to model on Figure 6 is shown on Figure 7. The subcomponents are enclosed inside context **DVMSystemSubCTX**, the inner component FSM is enclosed in **DVMSystemFSMCTX**, and the message handler is enclosed inside context **DVMSystem ShutdownMessageCTX**. Although DVM handles only the "Shutdown" message, it will also route the message "Initialize" through all its subcomponents (also DVM can send Initialize messages), therefore methods and gates for these two messages were generated and included in all contexts and router object.

Every (H)ALL component will have the following structure in CO-OPN:

– As was referred, for each component a context will be created and added to the module set of the target CO-OPN package.

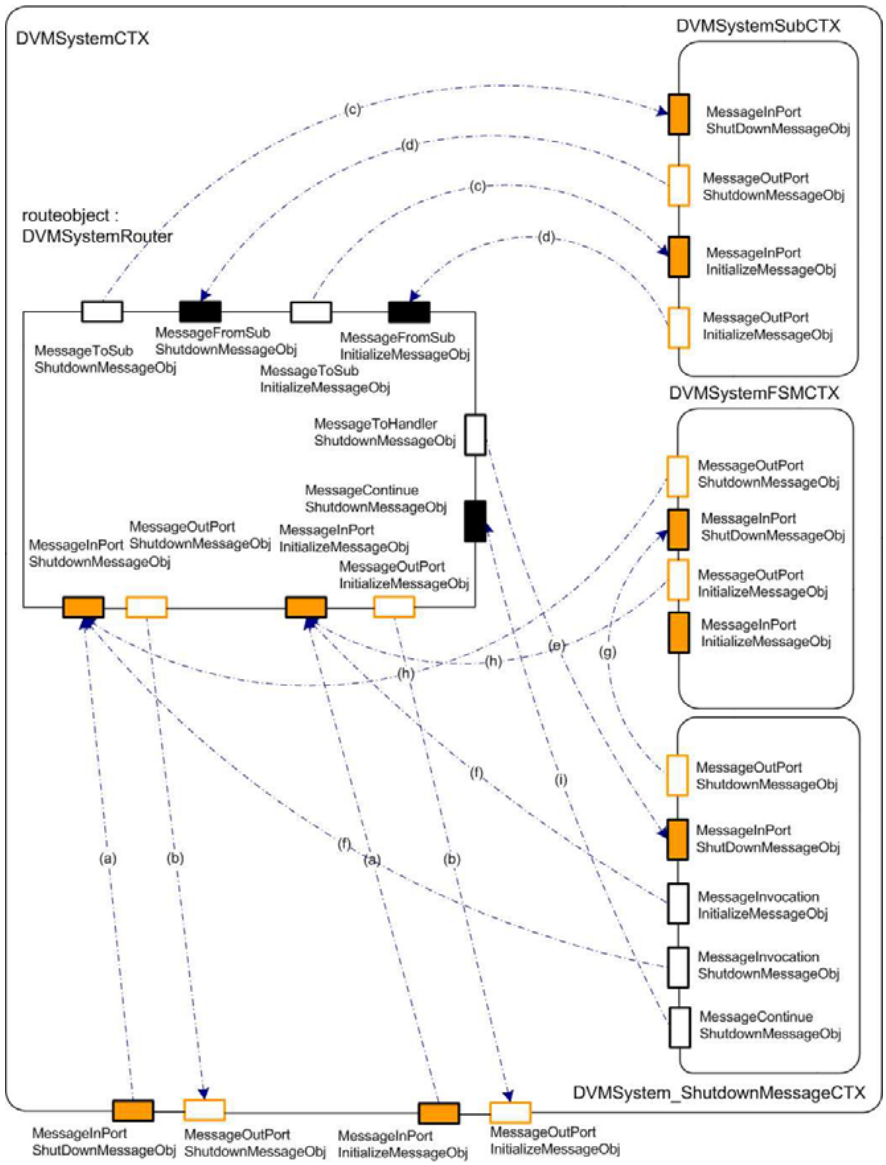


Fig. 7. DVM system component expressed in CO-OPN

– each component context will include a router as a CO-OPN object. The router object will route both commands and messages in and out of the context.

– each component context will include a FSM context, where the finite state machine logic of the component will be mapped.

- each component context will include (*context-use*) a SubComponents context. This context only encapsulates all subcomponents contexts and passes messages and commands in and out from them.
- each component context will include several message handler contexts. These contexts will have the finite state machine logic of the message handlers defined on (H)ALL's component.

The router object is implemented with a CO-OPN class. A pair of places *RouterEnabled* and *RouterDisabled* are added to class *Route*. Command *Enable* can also take effect on message handlers, thus for each component message handler, another pair of places ("Enabled" and "Disabled") are added to the Route class. All Enabled places are marked with "@" on Initial. Then some axioms switches the tokens between Enabled and Disabled places according to the value on the Enable command. Message routing to and from subcomponents context, as well as message routing from messages which were handled in some message handler are also implemented using axioms. The route object also routes commands. Axioms route commands from parents, returns from children to parents, and commands and returns to its inner component context.

4 Conclusions

We have described the syntax and the semantics of a language named (H)ALL for interface modeling. It was introduced under the scope of the project BATIC3S which will derive a comprehensive methodology for semi-automated, formal model-based generation of effective, reliable and adaptive 3D GUIs for diagnosing Control Systems. Although initially meant for 3D GUIs, (H)ALL has support for the definition of both 2D and 3D abstract widgets to compose either 2D or 3D GUIs. Similarly, the domain of application is not confined to control systems, but also for other application domains as long as they do share the same conceptual properties: hierarchical, event based, etc. Since the GUI library will be composed by graphical objects which are dependent from the used graphics-engine, and we did not wanted to bound our solution to a specific set of widgets, we have chosen not to include any other syntatic facility (meta-model) to enable the expression of anything more than their structure (hierarchical composition and data structures) and behaviour (state machines).

The next step of this project is to implement (H)ALL. This will be achieved by using meta-modeling tools and the corresponding transformation rules as defined in [2]. From the point of view of the architecture design, there will be two main modules: the graphical interface editor (to design and build the GUI prototypes) and the execution engine (to run and simulate the GUI prototypes). The next step will be to build an editor's prototype based on (H)ALL's definitions and its mapping rules to CO-OPN.

Another relevant issue to be tackled will be the need for optimising the mapping rules. Not only we will want to reduce redundant structures but also to improve the running speed of the coordination engine as well as the other engines.

This work represents a step forward closer to design a design framework with a visual edition pragmatics, which would allow for instance, free-hand editions of both structure and behaviour of remote user interfaces, without requiring all the interface system to be reset and rebuilt. With (H)ALL, we believe to be in the right direction to derive a proper design framework with a novel methodology that makes use of formal methods, languages and tools to express any model of a GUI.

References

1. Batic^{3}s. URL: http://smv.unige.ch/tiki-index.php?page=BATICS.
2. B. Barroca. (h)all: a dsl for rapid prototyping of user interfaces for control systems. Msc. thesis, Universidade Nova de Lisboa, 2007.
3. D. Buchs and N. Guelfi. A formal specification framework for object-oriented distributed systems. *IEEE Trans. Software Eng.*, 26(7):635–652, 2000.
4. B. collaboration. Batic^{3}s project - document collection 2005-2006. URL: http://smv.unige.ch/tiki-download_file.php?fileId=728.
5. F. Moussa, C. Kolski, and M. Riahi. A model based approach to semi-automated user interface generation for process control interactive applications. *Interacting with Computers*, 12(3):245–279, January 2000.
6. A. Puerta and J. Eisenstein. Towards a general computational framework for model-based interface development systems. In *IUI '99: Proceedings of the 4th international conference on Intelligent user interfaces*, pages 171–178, New York, NY, USA, 1999. ACM Press.
7. QVT-Partners. Qvt-partners. URL: http://qvtp.org.
8. M. Risoldi and V. Amaral. Towards a formal, model-based framework for control systems interaction prototyping. In *to be published at Springer Verlag Lecture Notes in Computer Science (LNCS) Series 2007*, 2007.
9. M. Risoldi and D. Buchs. A domain specific language and methodology for control systems gui specification, verification and prototyping. In *Proceedings of the 2007 IEEE Symposium on Visual Languages and Human Centric Computing, IEEE (to appear)*, 2007.
10. J. Vanderdonckt. Knowledge-based systems for automated user interface generation: the trident experience, 1995.
11. J. Vanderdonckt, C. Chieu, L. Bouillon, and D. Trevisan. Model-based design, generation, and evaluation of virtual user interfaces. In *Web3D Symposium Proceedings*, pages 51–60, 2004.

System-Model-based Simulation of UML Models

María Victoria Cengarle[1], Juergen Dingel[2,3], Hans Grönniger[3], and Bernhard Rumpe[3]

[1] Institut für Informatik, Technische Universität München, Germany
[2] Applied Formal Methods Group, Queen's University, Canada
[3] Institut für Software Systems Engineering, Technische Universität Braunschweig, Germany

Abstract. Previous work has presented our ongoing efforts to define a "reference semantics" for the UML, that is, a mathematically defined system model that is envisaged to cover all of the UML eventually, and that also carefully avoids the introduction of any unwarranted restrictions or biases. Due to the use of underspecification, the system model is not executable. This paper shows how the system model can serve as the basis for a highly customizable execution and simulation environment for the UML. The design and implementation of a prototype of such an environment is described and its use for the experimentation with different semantic variation points is illustrated.

1 Introduction

Modeling is a major activity in any development of a complex system. In contrast to all other engineering disciplines, software engineering still suffers from a distinct lack of maturity. Its foundations are still in development and discussion. The UML [1] is still undergoing heavy changes and criticism and we can expect that it will take a while until we have settled and generally agreed upon our foundations for modeling and specifying software intensive systems.

The modeling of software can serve quite a number of purposes within a development project. Commonly, models are used as high-level programming languages for generating code. Code generators and other tools for the UML do not necessarily agree on the generated structural and behavioral semantics, impeding interoperability and making the results of code generation, analysis or simulation tool dependent. This problem is rooted in one of the common criticisms of the UML, namely its lack of a precisely defined, commonly agreed semantics.

Work in [2] describes an ongoing effort to solve this problem by providing such a semantics in form of a mathematically defined *system model*, a precisely defined syntax of the UML and most importantly an explicit and unambiguously defined semantics mapping from UML syntax into the system model. This system model is defined using standard mathematical techniques while carefully avoiding any restrictions or biases that are not warranted by the UML standard.

This paper describes our work on using the system model as the basis for an execution and simulation environment. Encoding a declarative mathematical specification into an executable model requires design decisions. While some of these decisions are uncritical, such as the encoding of the universe of object identifiers as integers and variable names as strings, other decisions are more significant such as the realization of associations and the scheduling of object computation. The UML2 standard identifies many of these significant design decisions as *semantic variation points*. However, due to the informal nature of the UML specification, it is likely that the set of variation points is not comprehensive. Moreover, it is unclear what exactly good or bad choices (or values) for these variation points are, and if some of these choices interact with each other in negative or unexpected ways. While the UML2 standard sketches properties of a model of computation (see, e.g., [1, Section 6.3]) for UML models, the precise shape of that model together with its strengths and weaknesses does not become clear. The main goals of our execution and simulation environment are thus to allow us to

1. validate parts of the system model and to improve it,
2. gain a better understanding of exactly what kind of information is needed to make a UML model executable and how that information can be provided to an execution and simulation environment in a modular way that supports customization and experimentation,
3. experiment with different choices for semantic variation points, to determine "good" and "bad" choices, and uncover possible negative interactions and exclusions,
4. gain more experience with the model of computation envisioned in the UML2 standard.

While the prototype described in this paper does not achieve all of these goals yet, the results obtained are encouraging.

In the rest of the paper, we will briefly review the most relevant concepts of the system model as defined in [3–5] in Sect. 2. Then, the most interesting aspects of the design and implementation of the execution and simulation environment are described in Sect. 3. Finally, the use of the environment is illustrated in Sect. 4 by executing a given model under different scheduling strategies and observing the resulting differences in behavior. Sect. 5 sketches related work and concludes the paper.

2 Overview of the Theory of System Models

The goal of system models is to provide a solid mathematical basis for the definition of formal semantics of modeling languages, in particular for the UML. A system model describes the universe (set) of all possible semantic structures, each with its own behavior, of the system defined by a sentence of the syntax. Such a sentence is, in the case of UML, a complete UML specification.

Roughly speaking, a system model is a timed state transition systems, whose states embody the appropriate data structures and whose transitions mirror the behavior as defined by the sentence of interest. In order to define these kind of timed state transition systems, we made use of a combination of theories that deal with data, objects, classes, control, messages, calls, returns, recursive invocation, events, threads, time, scheduling, among other concepts.

The current state of the realization presented in Sect. 3 slightly simplifies the definition reported in [3–5]. The summary of the present section, thus, only addresses the relevant parts of the system model definition as used in the next section.

Different principles guided the definition of the theory of system models. Instead of using a specialized notation such as, e.g., Z, B, or ASMs, we relied on mathematics directly, in particular on the theories of numbers, sets, relations, and functions. Those other notations are biased towards certain purposes as for instance model checking analysis and thus are less versatile.

The theory of system models does not constructively define its constituents, i.e., its elements are descriptively characterized. Moreover, no implicit assumptions are made: what is not explicitly specified, need not hold. In addition, the theory of system models presents various so-called semantic variation points, that allow a specialization of the theory to particular realms. Other principles that guided the definition of system models can be found in [3–5].

A system model is a timed state transition system. States of a system model are composed by static information (basically values and store), dynamic information (basically threads and control), and a finite number of message pools (in the simplest case, queues of incoming messages like method invocation and signals). A data store is a partial function

$$\mathsf{DataStore} : \mathsf{UOID} \hookrightarrow \mathsf{UVAL}$$

that maps object identifiers onto values. DATASTORE denotes the universe of all possible data stores. CONTROLSTORE denotes the universe of all possible control states. A control state contains information about all the threads being executed:

$$\mathsf{ControlStore} : \mathsf{UOID} \hookrightarrow \mathsf{UTHREAD} \hookrightarrow \mathit{Stack}(\mathsf{UFRAME}),$$

where the universe of execution frames is defined by

$$\mathsf{UFRAME} = \mathsf{UOID} \times \mathsf{UOPN} \times \mathsf{UVAL} \times \mathsf{UVAL} \times \mathsf{UPC} \times \mathsf{UOID}.$$

That is, a frame contains the identifier of the executing object, the operation invoked, parameter values packed in a record, values of the local variables likewise packed in a record, the program counter, and the identifier of the caller object. Finally, UEVENT denotes the universe of possible events. The events of an object are retrieved via the function

$$\mathsf{EventStore} : \mathsf{UOID} \to \wp(\mathsf{UEVENT}).$$

The universe of all possible event stores is denoted by EVENTSTORE.

States of a system model are thus defined as triples (ds, cs, es), where $ds \in \mathsf{DATASTORE}$, $cs \in \mathsf{CONTROLSTORE}$, and $es \in \mathsf{EVENTSTORE}$. How the three stores are used to capture the state of a single object and the system is explained in more detail in Sect. 3.

A *system model* is a tuple $(\mathsf{STATE}, \Delta, \mathsf{Input}, \mathsf{Output}, \mathsf{Init})$ with

- STATE a set of states of the form (ds, cs, es) as described above,
- Input and Output the input and output channel sets, respectively,
- $\mathsf{Init} \subseteq \mathsf{STATE}$, and
- $\Delta : (\mathsf{STATE} \times \mathsf{T}(\mathsf{Input})) \to \wp(\mathsf{STATE} \times \mathsf{T}(\mathsf{Output}))$ a transition function

where $\mathsf{T}(C)$ is the set of possible channel traces for the channel set C, i.e., if $x \in \mathsf{T}(C)$ and $c \in C$, then by $x.c$ we denote the finite sequence of messages $x(c) \in \mathsf{UMSG}^*$ where UMSG is the universe of all messages.

Timed state transition systems behave like a Moore machine,[4] that is, the output depends on the state and not on the input. Moreover, the state transition function is total. The theory of system model encompasses further definitions, like composition of system models and interface abstraction. These provide comfortable tools that allow compositional definition and abstraction from representation details. They moreover fulfill interesting and desirable properties. Due to lack of space, the interested reader is referred to [3–6].

Composition allows to independently implement parts of a specification and subsequently compose these; the system model this way obtained implements the complete UML specification. The virtual machine presented in Sect. 3 below implements the state transition function. This implementation does not reflect this abstract view on the composed system directly. Instead, a fine-grained account on control flow and the effect of actions on an object's state is given.

3 Design and Implementation of a System Model Simulator

This section provides an overview of the current state of an implementation of the system model in Haskell. A basic knowledge of Haskell is assumed. For more information see [7, 8].

The current state of the implementation differs from the system model definitions given in Sect. 2 and in [3–5] in various ways: Some simplifications have been made that will, however, be brought in line with the mathematical definitions as the implementation advances. Furthermore, the mathematical definitions allowed us to leave quite a number of execution relevant issues unspecified that have to be considered in order to make the system model executable. In general, the predicative specification style is replaced by an executable simulator that allows for experimentation with different choices for variation points. Just like in the declarative system model, we tried to avoid unnecessary overspecification. Whenever design decisions were necessary to achieve executability, we

[4] A Moore machine is a finite state machine that produces an output for each state.

implemented these in a modular, replaceable way. We, e.g., had to provide constructive solutions for scheduling or an action language. In developing an action language we tried to stay in line with the UML standard in the sense "that all behavior in a modeled system is ultimately caused by actions", cf. [1, Sect. 6.3.1].

An important decision to make is, which language to use for implementing the simulation engine. We decided against the natural choice of using an object-oriented language, but use the lazy functional language Haskell for several reasons. First, Haskell is more similar to mathematics and therefore allows us to stay closer to the mathematical definitions. Second, with Haskell very compact specifications can be defined. And third, constant confusion arises if the encoding language is conceptually close to the encoded language, i.e., it is always yet to clarify whether it is a simulated class or a class used to handle the simulation. Haskell is conceptually distant enough from the UML so that such problems do not occur. Furthermore, this conceptual distance enforces a deep embedding of the UML into Haskell. For example the typing system of the UML is independent from the Haskell type system and is completely encoded within a universe of types on its own.

In Sect. 3.1 we first explain the architecture of the system model simulator and highlight the main concepts necessary for this paper from each module. Data needed to set up the execution and the several execution steps themselves are described in Sect. 3.2. Finally, the currently realized choices for variation points of the system model are discussed in Sect. 3.3.

3.1 Architecture

The architecture of the implementation has two conceptual layers. First, the system model is implemented by a set of Haskell modules and provided to the simulator by a dedicated module *SystemModel*. The system model part of the architecture again is built in a layered form where sophisticated concepts depend on more fundamental definitions. The system model modules mirror the definitions from the mathematical version but are extended to incorporate functions needed for executability. Fig. 1 shows the architecture as a set of dependent Haskell modules. Modules *Map*, *Stack*, and *Buffer* are general purpose implementations with the obvious operations.

Basics *StructureBasics* defines structure-related system model entities. The universe of type names, for example, is defined as a data type

```
UTYPE = TInt | TVoid | .. | XClass UCLASS
```

where `UCLASS` is a class definition. "Primitive" types as `TInt` start with a `T`. Because `UCLASS` (universe of class names) mathematically is a subset of `UTYPE` but defined on its own, it is wrapped by the type constructor `XClass` to become part of `UTYPE`. The universe of values `UVAL` introduces type constructors which start with a `V`, e.g. `VInt 2` represents the integer value 2 and `VVoid` the singleton value "void". The universe of object identifiers `UOID` is integrated into `UVAL` by

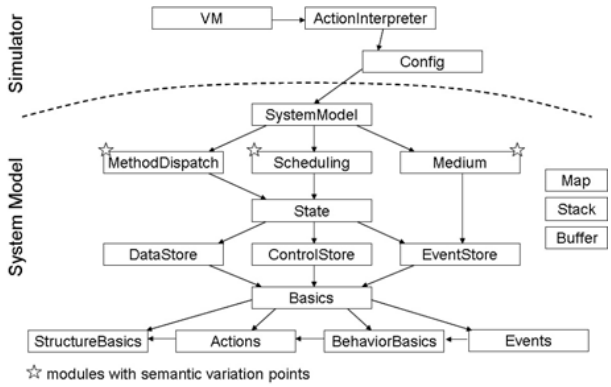


Fig. 1. Architecture of the system model in Haskell

introducing the type constructor `XOID`. Further, this module provides a definition of a subclassing relation as `SubclassRel = Map UCLASS [UCLASS]` modeling multiple inheritance, if needed. `UOPN` defines the signature of a method and is defined in *StructureBasics*, while `UMETH`, which among other things contains a list of actions that constitute the method's body, is defined in *BehaviorBasics*. The mapping `MethMap` records for each class C the operations contained in C and which methods implement each of these operations. Module *BehaviorBasics* also contains definitions of `UFRAME` and `UTHREAD` which roughly coincide with the mathematical definitions.

Module *Actions* contains a data type that defines the action language that is interpreted by the *ActionInterpreter*. It contains actions for sending messages, creating objects, setting attributes or variables and for doing basic arithmetic. As an example, actions will be used in Sect. 4 to provide an implementation for a method. Events defined in module *Events* are `CallEvent`, `ReturnEvent` and `SignalEvent` each associated with a message of type `UMSG`. The events are used to trigger synchronous operation invocation, operation return or asynchronous communication, respectively. For convenience, the list of basic modules is collected in a module *Basics*.

State The state of the system model (module *State*) is determined by the three stores, data, control, and event store, implemented in the respective modules. So, the state is of type `USTATE = (DataStore, ControlStore, EventStore)`. Stores are equipped with functions to retrieve the current state of an object (attribute values, currently executing threads, and pending events). Additionally, functions are provided to construct empty stores, or to add or update state information for a given object.

SystemModel & Simulator All interaction between objects (asynchronous communication as well as method invocation) is achieved by the generation

of events that lead to messages passed from one object to the event buffer of another. The module *Medium* provides a function that "transmits" messages between (potentially physically) distributed objects over some communication media. The concrete implementation of the medium (e.g. lossy or reliable transmission) is a semantic variation point (cf. Sect. 3.3) and can easily be replaced in the simulator. Another variation point is given by the way how method calls are resolved. Various approaches exist (e.g. single or multiple dispatch) and can all be used by implementing a method `MethodDispatcher` in module *MethodDispatch* appropriately.

The way computation is scheduled in the system model is largely left open in the mathematical definitions. In the implementation, the required function signatures have been fixed in the module *Scheduling*. Concrete scheduling strategies can be realized as a further variation point.

The described modules are provided to the simulator through the module *SystemModel*. The simulator consists of three modules. *ActionInterpreter* interprets each basic action while the virtual machine *VM* schedules the next object and thread for execution based on the system model configuration *Config*.

3.2 Execution

This section provides a more detailed description of the basic steps that constitute a simulator run.

Input to Simulator In order to run the simulator, it has to be provided with three kinds of data:

1. Decisions on specific variation points in the system model (e.g., decisions about the kind of scheduling, method dispatching etc.).
2. Description of the system to be simulated in terms of the system model (definitions of classes, methods etc.).
3. Initial setup, that is, the state of the system in which execution starts (generally, a network of initially created objects that may start executing an operation).

The selection of variation points and the system description will be used throughout the simulation and is handed over to the simulator in form of a data type `Config` together with the following functions

- `dispatcherOf :: Config -> MethodDispatcher`
- `schedulerOf :: Config -> (RunnablesSel, Scheduler)`
- `mediumOf :: Config -> Medium`
- `subclassRelOf :: Config -> SubclassRel`
- `methMapOf :: Config -> MethMap`

The first three functions return the specific choices of the user for the different variation points concerning method dispatching, scheduling, and the type of medium.

The specification of the simulated system is typically provided in a separate Haskell module that contains class and method definitions, and a definition of `Config`. Additionally, the initial setup may be specified as a type

```
Setup = [(String, UCLASS, OKind, [String])]
```

In the setup it is decided whether an initial object should be active or passive. The data type `OKind = Active UOPN Prio | Passive` either contains an operation and a priority or it indicates that the object should be passive. The setup is used as input for function `runMain` of the virtual machine that constructs the initial state. For each entry in the setup list `(a,cl,k,cons)` an object of type `cl` is created, either starting execution in operation `op` with priority `prio` if `k` matches the pattern `Active op prio` or remaining passive, and having a link to each of the other objects in list `cons`. The objects at this point are identified via a string `a` because their actual object identifier values are not yet known. Usage of `runMain` is optional as the initial state may also be created manually in order to start the simulator via the function `run`.

Top-level loop of VM Module *VM* implements the following function signatures to start execution:

```
runMain :: (Config, Setup) -> (USTATE, Time)
run :: ((Map ThreadID Time), Time, Config, USTATE) -> (USTATE, Time)
```

Function `runMain` constructs the initial state (of type `USTATE`) and calls `run`. After the simulation is completed the final state and time is returned. The function `run` forms the "main loop" of the simulation and is called recursively at the end of each step. It takes two additional arguments. The first parameter is a map that stores the last execution time for each thread and the second provides a notion of the current execution time (which counts the number of simulation steps). The Haskell code in Fig. 2 summarizes the behavior of `run`.

```
1 run (times, t, conf, state) =
2   let (rselector, schedule) = schedulerOf (conf)
3       runnables = collectRunnables (rselector, state)
4       runnables' = addLastExecInfo (times, runnables)
5       (oid, thread) = schedule (t, runnables')
6       state' = exec (oid, thread, state, conf)
7       times' = update (times, (thread, t))
8   in if (null runnables) then (state, t)
9      else run (times', (t+1), conf, state')
```

Fig. 2. Haskell code for function `run`

The process of scheduling the next object and thread is done in two steps. First, given an object and a state, a function `rselector` collects the threads of the object that are ready for execution. In line 2, the type of `rselector` is

```
RunnablesSel = (USTATE, UOID) -> [(ThreadID, Prio)]
```

For each object the information available to decide which threads to offer for scheduling is the whole state of the object. In particular, this function is able to inspect the object's event buffer to check whether newly arrived events need to be processed. The function called in line 3 is a higher-order function defined in module *Scheduling* of signature

```
collectRunnables :: (RunnablesSel, USTATE) -> [(UOID, ThreadID, Prio)]
```

that collects the information about runnable threads for each existing object in the data store using the function given as the first argument. If the list of runnables is empty, the execution ends and the final state is returned (line 8). Otherwise, the list is extended with the last execution times of each thread after which it is of type `[(UOID, ThreadID, Prio, Time)]` (line 4). Now, the second part of the scheduling process takes place. Function `schedule` of type

```
Scheduler = (Int, [(UOID, ThreadID, Prio, Time)]) -> (UOID, ThreadID)
```

schedules the next object and thread combination.

The next atomic execution step is carried out in line 6 using the function `exec` defined in module *VM*, and will be explained with the help of the Haskell code given in Fig. 3 below. Finally, the map with the last execution times is updated and `run` is called recursively (lines 7 and 9).

```
1  exec :: (UOID, ThreadID, USTATE, Config) -> USTATE
2  exec (oid, threadid, state, conf) =
3    let state' = consumeEvent (state, conf, oid, threadid)
4        cs = csOf (state')        -- get the controlstore from state
5        thread = threadOf (cs, oid, threadid) -- get the thread
6        md = dispatcherOf (conf)
7        scl = subclassRelOf (conf)
8        mmap = methMapOf (conf)
9        ds = dsOf (state')        -- get the datastore from state
10       op = opOf (thread)        -- get the operation from thread
11       (Meth _ _ actions) = md (scl, mmap, ds, oid, op)
12       pc = pcOf (thread)        -- get the pc from thread
13       action = actions!!pc
14       state'' = interpret (action, state', thread, conf)
15   in state''
```

Fig. 3. Haskell code for function `exec`

Carrying out an atomic step The function `consumeEvent` called in line 3 of function `exec` (Fig. 3) removes an incoming event (if any) from the object's

buffer and prepares the context for handling this event. In case of a method call, it constructs a thread frame containing all relevant information (parameter values, etc.); in case of a return event it restores the thread's context and provides the return value as a local variable. For an asynchronously received event a thread is created that starts executing the operation stated in the event's message. In lines 4-10, information used for method dispatching is extracted from the configuration and state. The method dispatcher `md` is a function with signature

```
MethodDispatcher = (SubclassRel, MethMap, DataStore, UOID, UPON) -> UMETH
```

Based on the system description (subclassing and relationship between classes, operations and methods), the data store (used to retrieve the type of the object), the object identifier and the operation to call, the actual method is determined and returned. The returned method contains a list of actions (line 11). The next action to execute is pointed to by the current value of the program counter of the thread. Interpretation of the action is done by calling the function `interpret` of the module *ActionInterpreter*. For each action defined in module *Actions* there exists a corresponding part of the interpreter function that is selected by the pattern matching mechanism of Haskell. The signature is

```
interpret :: (Action, USTATE, UTHREAD, Config) -> USTATE
```

In general, this function makes changes to the system's state (by altering attribute values, creating events, adjusting the program counter, etc.). If the action involves sending of an event to another object (e.g. in case of an operation call), `interpret` accesses the medium using the function `mediumOf` on the `Config`. The function `Medium = (EventStore, Event) -> EventStore` is responsible for putting the event in the right event buffer of the receiving object.

3.3 Variation Points

Variation points in the system model implementation mark points where different realization variants of a function may lead to different behavior or efficiency of the simulator and therefore the simulated system. To allow experimentation, the Haskell implementation of the system model provides different alternative realizations of variation points that can be exchanged in different runs of the simulator. Currently, the following variation points are considered in the implementation explicitly:

1. Scheduling
2. Method dispatching
3. Event distribution by a medium

Selecting implementation alternatives for these tasks is possible by using the data type `Config` in different combinations. In the future, other variation points mentioned in [3–5] will be added, for example, different realizations of associations in the system model which have not yet been considered in the implementation. Facilitating alternative structures or behavior is also possible

by replacing one or more modules by alternative implementations. This, for example, allows for adding additional basic types to the UTYPE structure.

Up to now, we have only started to explore the scheduling variation point in greater detail. For the other two variation points single implementations are provided. Method dispatch (supporting single inheritance) is done by looking up a method in the class of the object; if not found there, the method is looked for in the superclass(es). The medium is implemented as a "reliable" medium that without loss or reordering transfers an event into the receiver's event buffer.

Scheduling Recall that scheduling is performed in two steps. First, a function of type `RunnablesSel` is used to retrieve a list of runnable treads for each object. Two alternative implementations are provided:

RTC The function does not consider any event in the buffer while another thread is still active in the object. This is also known as *run-to-completion* execution.

CONC As soon as an event is put in the event buffer, the function also offers the thread that would handle this event to the scheduler, leading to potentially *concurrent* execution of multiple threads in one object.

Second, based on the current list of runnable threads, one thread in one object is scheduled for execution. Here, also two variants of scheduling strategies have been implemented:

RR Round-robin scheduling selects all threads alternately.

PRIO A priority-based scheduling finds the thread with the highest priority and smallest last execution time and selects it for execution. Priorities change dynamically in that the effective priority is computed by a thread's base priority plus its waiting time (aging).

Although we provide a single function for scheduling all objects, the implementation of more complex scheduling strategies that make use of different scheduling domains and different "sub schedulers" (e.g. to handle groups of objects that live on the same processing node) is also possible.

4 Example

The purpose of this section is to show how a concrete system can be described in terms of the system model and how to configure and run the simulator with different values for the choices for the variation points.

The system we are going to simulate is depicted in Fig. 4. The *Producer* is an active object executing its *produce* method that produces two data elements (in this example the integer values 10 and 20 will be produced) and provides them by calling method *put* on the buffer before it terminates. We assume that producing a new data element takes a long time. This is modeled by "counting to 5" each time before making one data element available. Method *put* sets the

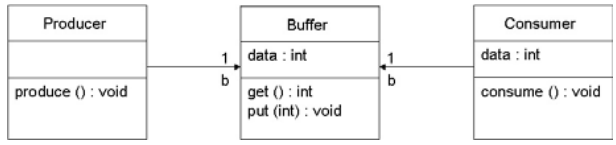


Fig. 4. Example system

attribute *data* to the received value, overwriting the stored value whether or not it has been fetched before. A *Consumer* is also an active object that tries to fetch exactly one data element from the buffer. It continuously calls *get* which returns -1 if no data is available. After successfully getting a value from the buffer, the consumer sets its *data* attribute to that value and terminates. The method *get* sets the buffer's attribute *data* to -1 if a consumer has fetched the current value so that no other consumer will get the same value.

Despite its simplicity, this example suffices to show the effects of the different variation points when running the simulator.

Due to lack of space, we only show how the class *Buffer* and its *put* method are described in terms of the system model. In Fig. 5, lines 1-2 show a class declaration (constructor `MClass`), the attribute of a class `buffer` is kept in a record (constructor `MRec`), is called `data` and is of type `TInt`. In lines 4-5 operation `put` is declared. It takes a single parameter of type `TInt` and its return type is `TVoid`. The definition of a method implementing operation `put` starts in lines 7-8 by stating that it implements operation `putOp`, that the parameter name is `p` and that the method's body is given as a list of actions called `putActions`. In line 10 a local variable of name `d` with type `TInt` and initial value zero is created. The following action assigns the value of the parameter `p` to the local variable `d`. The data attribute is set in line 12 and finally `VVoid` is returned.

```
1  buffer :: UCLASS
2  buffer = MClass "Buffer"  (MRec [("data", TInt)])
3
4  putOp :: UOPN
5  putOp = MOp "put" [TInt] TVoid
6
7  putMeth :: UMETH
8  putMeth = MMeth putOp [("p",TInt)] putActions
9
10 putActions = [CreateLocalVariableAction "d" TInt (VInt 0)
11              ,SetLocalVariableFromParamAction "d" "p"
12              ,SetAttribAction "data" "d"
13              ,SendReturnAction (VVoid)]
```

Fig. 5. Class and method definition for the Buffer

In the same way, the producer class `prod`, the consumer class `cons` and their operations and methods are declared. The next step in the system description is to connect classes, operations and methods. To this end, `methMap` maps class `buffer` to the map that defines that operation `putOp` is implemented by `putMeth` and `getOp` by `getMeth`, and maps classes producer and consumer to operations `prodOp`, `consOp` implemented in `prodMeth` and `consMeth` respectively. In this example we do not have subclassing so we define the subclassing relation to be the empty map `subc = emptyM`.

The initial setup in which execution should start is shown in Fig. 6. The initial system state will have one passive object of type `buffer` (identified for further reference by `"b"`) and three active objects: two consumer objects `"cons1"` and `"cons2"` execute operation `consOp` with priority 1, and having a link to `"b"`. Also, one producer `"prod1"` executing operation `prodOp` with higher priority 10 is created. With the help of a framework for domain specific languages [9], all Haskell definitions have been generated from textual versions of UML's class diagrams (with lists of actions for method implementations) and object diagrams (for the initial setup).

```
setup = [("b"    , buffer, Passive,          []),
         ("cons1", cons,   Active consOp 1,  ["b"]),
         ("cons2", cons,   Active consOp 1,  ["b"]),
         ("prod1", prod,   Active prodOp 10, ["b"])]
```

Fig. 6. Initial setup for simulation

The first concrete configuration `conf` of type `Config` with which the simulator is called uses standard implementations for method dispatching and medium as discussed in Sect. 3.3. For the scheduling strategy, we'll start with concurrent execution of threads in one object (CONC) and a round-robin scheduling (RR). Calling function `runMain (conf, setup)` computes a final system state. Fig. 7 shows only the console output of the data store of the final state with the attributes and values for each object. It can be seen that the two consumer objects both received the same value "10" and the second value "20" still remains in the buffer. The concurrent execution of the method `get` thus led to inconsistencies illustrating the fact that the buffer code is not thread-safe (in a different setup other inconsistencies, e.g., due to concurrent `put` and `get` are possible, too). The total number of steps needed for simulation is 221.

For the second run we use run-to-completion execution (RTC) and priority-based thread scheduling (PRIO). The resulting data store is depicted in Fig. 8. The buffer in the final state is empty and each consumer received one of the two produced values. Since only one thread is allowed to "enter" the object `buffer`, no inconsistencies are possible. Moreover, the number of steps required is reduced to 64% compared to the previous run because the computation intensive production of data elements in the producer received a higher priority. The two

```
attributes:
Producer(id 0): [("b",XOID 3)]
Consumer(id 1): [("data",VInt 10)
                ,("b",XOID 3)]
Consumer(id 2): [("data",VInt 10)
                ,("b",XOID 3)]
Buffer(id 3): [("data",VInt 20)]
time: 221
```

Fig. 7. Result with concurrent threads and round-robin scheduling

```
attributes:
Producer(id 0): [("b",XOID 3)]
Consumer(id 1): [("data",VInt 20)
                ,("b",XOID 3)]
Consumer(id 2): [("data",VInt 10)
                ,("b",XOID 3)]
Buffer(id 3): [("data",VInt -1)]
time: 142
```

Fig. 8. Result with run-to-completion and priority-based scheduling

other combinations, (CONC + PRIO) and (RTC + RR), lead to faster execution with inconsistent results or to slower execution with correct results, respectively.

5 Conclusion

In this paper we described the implementation of a virtual machine for the simulation of UML-specified systems. The virtual machine does not interpret UML models directly, instead UML models can be mapped into the system model that is then simulated under consideration of the specific choices for the variation points. The development of the simulator allows us on the one hand to validate the system model given in [3–5] and on the other to experiment with different choices for variation points.

Our work differs from the work in [10] in that we also support the description of behavior while [10] focuses on structural modeling. The virtual machine described in [11] focuses on traceability of models. UML class diagrams and sequence diagrams are supported. Code generation is parameterized to some extent. In the industrial area various commercial tools exist that all implement a certain executable subset of UML with a fixed semantics, e.g. [12]. The work closest to ours is that of [13] in which a generic model execution engine is used as a basis for a UML simulator that also supports semantic variation points. Moreover, there exists a wealth of related work regarding the definition of UML semantics. Most approaches, however, focus on the semantics of one or two diagrams (e.g. [14–16]) whereas we aim at defining semantics for a larger set of diagrams. Yet, a detailed discussion of related work in that area is beyond the scope of this paper.

The development of our simulator is still ongoing. Some concepts from the mathematical definitions still have to be integrated (e.g. associations) and support for more variation points has to be added. Further, the basic actions and execution mechanisms may also be compared and aligned with the results of the OMG's standardization of Executable UML [17]. As a next step, it is planned to systematize the actual mapping from UML diagrams to the system model and to identify additional variation points in the mapping.

Acknowledgement: The work presented in this paper is undertaken as a part of the MODELPLEX project. MODELPLEX is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (02- 06). Information included in this document reflects only the authors' views. The European Community is not liable for any use that may be made of the information contained herein. This research is supported by the DFG as part of the rUML project.

References

1. Object Management Group: Unified Modeling Language: Superstructure, version 2.1.1 (07-02-03) (February 2007)
2. Broy, M., Crane, M., Dingel, J., Hartman, A., Rumpe, B., Selic, B.: 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In: MoDELS 2006 Workshops, Lecture Notes in Computer Science 4364. (2006) 318–323
3. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, Institut für Informatik, Technische Universität München (June 2006)
4. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The Control Model. Technical Report TUM-I0710, Institut für Informatik, Technische Universität München (February 2007)
5. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, Institut für Informatik, Technische Universität München (February 2007)
6. Cengarle, M.V.: System model for UML – The interactions case. In: Methods for Modelling Software Systems (MMOSS), Dagstuhl, Germany (2007)
7. Thompson, S.: Haskell: The Craft of Functional Programming. Second edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
8. Bird, R.: Introduction to Functional Programming using Haskell. Second edn. Prentice Hall Series in Computer Science. Prentice Hall (1998)
9. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. Accepted for MODELS 2007 (2007)
10. Riehle, D., Fraleigh, S., Bucka-Lassen, D., Omorogbe, N.: The architecture of a UML virtual machine. In: Proc. OOPSLA '01. (2001) 327–341
11. Wada, H., Babu, E.M.M., Malinowski, A., Suzuki, J., Oba, K.: Design and Implementation of the Matilda Distributed UML Virtual Machine. In: Proc. of the 10th IASTED Intl. Conf. on Software Eng. and App., Dallas (2006)
12. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc. (2002)
13. Kirshin, A., Dotan, D., Hartman, A.: A UML Simulator Based On A Generic Model Execution Engine. In: Proc. of the 20TH EUROPEAN Conference on Modelling and Simulation (ECMS 2006), Bonn, Germany (2006)
14. Latella, D., Majzik, I., Massink, M.: Towards a formal operational semantics of UML statechart diagrams. In: Proc. FMOODS'99, Kluwer (1999)
15. Kuske, S., Gogolla, M., Kollmann, R., Kreowski, H.J.: An integrated semantics for UML class, object and state diagrams based on graph transformation. In: IFM '02: Proc. of the Third Intl. Conf. on Integrated Formal Methods. (2002)
16. Störrle, H.: Semantics of Interactions in UML 2.0. In: VLFM'03 Intl. Ws. Visual Languages and Formal Methods, at HCC'03. (2003)
17. Object Management Group: Semantics for a foundational subset for executable UML models, request for proposal ad/2005-04-02 (April 2005)

Model-driven Approach for Interactive Configuration Description in Component-based Software Product-lines

Juha-Matti Vanhatupa[1], Imed Hammouda[1], Mika Korhonen[2], Kai Koskimies [1]

[1]Institute of Software Systems, Tampere University of Technology,
P.O.BOX 553 FI-33101 Tampere, Finland
[2]John Deere Forestry Oy, PL 474, 33101 Tampere, Finland
juha.vanhatupa@tut.fi, imed.hammouda@tut.fi, KorhonenMikaP@JohnDeere.com,
kai.koskimies@tut.fi

Abstract. A current trend in component-based software product-lines is to specify individual products in XML-based configuration files. While this approach offers greater flexibility to product derivation, it usually results in a more complex derivation process due to the low level nature of XML description, large size of configuration files, and hidden dependencies between related configuration parts. Any slight error in the configuration file can lead to faulty products. This paper presents a model-driven approach to XML-based variation management of complex software platforms, aiming at more user friendly and controlled product derivation process. The proposed approach has been applied to build a guided configuration management environment for a harvester condition monitoring platform, based on extending an existing task-based framework specialization tool.

Keywords: Variation management, system configuration specification, modeling, XML

1 Introduction

Software product-lines [2,4] are a widely used approach to increase the productivity of software development. A product-line consists of reusable assets and an application development interface. New products are developed in the product-line by exploiting the variation provided by the application development interface. The form of the application development interface may vary from a simple API to a domain-specific language.

In many cases the reusable assets of a software product-line constitute a platform consisting of software components that are configured in different ways to achieve the functionality required by different products belonging to the product-line. This kind of product-line platform is black-box in the sense that the application developer is not expected to write new, application-specific code, but rather just define the values of certain properties of existing components and the way these components are connected. This kind of description may be given using a special-purpose language

rather than a programming language, which may further raise the abstraction level and relieve the application developer from the burden of knowing the details of the platform implementation. A natural technology for developing such a special-purpose language is XML; this choice makes immediately available a plethora of mature tools to create, analyze and process expressions given in the language. We call this kind of an XML document a *configuration description*.

However, although XML configuration descriptions are in principle human writable and readable – when assisted by proper tools – the specification of a product in XML can become very cumbersome and laborious. Typically, in real life a configuration description consists of hundreds of XML lines, with complicated implicit rules and dependencies constraining the legal descriptions. Since the rudimentary structure of XML does not allow intuitively appealing language constructs making explicit the variation points and their usage, the writing of a configuration description may require considerable expertise and carefulness, thus counteracting the idea of a high-level application development interface.

In this paper we propose a technique for alleviating this problem. We promote interactive support for creating a configuration description, rather than straight one-shot generation of a configuration description from a high-level representation. The main advantage of this approach is that the creator of a configuration description – the application developer – experiences the configuration description as a visible software artifact, the "source code" of the product, rather than as an internal file interesting only for the machine. The application developer is thus free to edit the configuration description in any way she wants, if needed. The tool only guides in the creation of a legal description (with respect to the rules of the platform), and gives warnings of violations.

The rules which specify how a configuration description is assumed to be given for a particular product-line platform are called *configuration patterns*. Our vision is model-driven in the sense that we assume a *configuration profile* which defines a simple metamodel for abstract configuration descriptions. The product-line architect is expected to provide a *configuration model* as an instance of this profile, while the application developer creates a concrete *configuration* as an instance of the configuration model. The tool set is eventually expected to generate appropriate configuration patterns as well as the XML schema on the basis of the configuration model. The configuration patterns are then used to assist the application developer in creating a valid configuration description, without knowing the platform architecture, the XML Schema structure, or the special rules related to the configuration descriptions of this particular platform. However, so far this vision is realized in our prototype tool only partially: the input for the tool is given directly as a schema, rather than as a configuration model, and an instance of the configuration model is only visualized in a graphical form. We have implemented the prototype by extending an existing Eclipse-based tool platform for pattern-based task-driven development [13]. The prototype tool has been evaluated in an industrial context by developing a configuration specification environment for a monitoring system of a harvester product family [16]. The problems encountered in the specification of the configuration files for this platform were the original motivation for this work.

The main contributions of this paper are (i) a general model-driven interactive approach for configuration description of product-lines, (ii) a (partial) realization of

this approach in a tool environment, and (iii) a realistic evaluation of the approach in industrial context.

We proceed as follows. In the next section we present our vision of a general model-driven approach for configuration-based variation management in product-lines in more detail. In Section 3 we briefly explain the generic pattern-driven tool environment used in this work. In Section 4 we discuss the current implementation of the approach, and in Section 5 we present the case study. Finally, related work is discussed in Section 6 before concluding remarks in Section 7.

2 Basic Approach

Our model-driven approach for configuration description of product-lines is based on three core elements: a configuration model for the product-line, an XML schema for the configuration, and a configuration pattern. Figure 1 depicts the elements of our approach. The two elements in the metamodel layer (meta product-line level) are domain independent: a profile for configuration models (configuration profile) and a metamodel for configuration patterns (pattern metamodel). The model layer components (product-line level), which are instances of their corresponding metamodel level elements, are domain (or platform) specific as their specification depends on the target domain of the product-line. The components in the instance layer (product level) are product specific. These are the elements accessed by a configuration tool.

Using a configuration tool, the user creates an XML-based configuration description for a specific product in the product-line. The role of the configuration pattern is to guide the application developer step-by-step in the creation of a configuration description providing task-driven interactive support. The pattern instance represents a valid configuration with respect to the platform rules. In this way, no illegal component configurations can be produced. We will discuss each of these elements in more detail in the next section following.

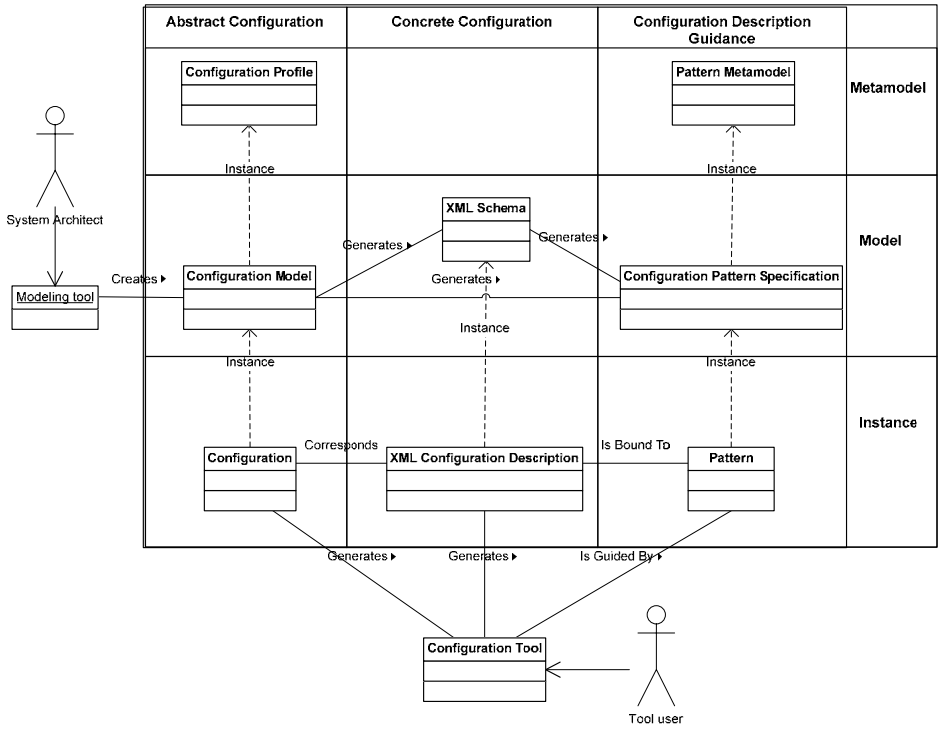


Fig. 1. Basic approach

2.1 Configuration Profile

A configuration profile represents a metamodel for configuration models, defining the main concepts and constraints in these models. Table 1 presents a set of stereotypes and constraints for the configuration profile we have used in our work. The profile defines *configuration* elements consisting of *component* and *attribute* elements. The configuration profile itself is very generic and simple, specifying configuration models as structural models consisting of components with possible subcomponents and attributes. In addition, components may depend on other components. In this context a dependency means that a component cannot exist in a configuration without another component. Attributes are typically valued in an instance of a configuration model (that is, in a particular configuration). Only the elements and relationships specified by the profile are allowed in a configuration model.

Table 1. Stereotypes of the configuration profile.

Stereotype	Base	Constraint
Configuration	Class	There must be exactly one Configuration element in the model. Configuration can have a composition relationship to any number of components.
Component	Class	Component may have dependency to any number other components. Component can contain any number of other components. Component can contain any number of attributes.
Attribute	Property	-

2.2 Configuration Model

A configuration model defines the structure of configurations given for products in the product-line. Because elements of the model level are domain-specific, we use a simple configuration model as an example (see Fig. 2). This configuration contains two kinds of components: *producers* and *consumers*. Producers have three attribute elements: *Quality*, *Address* and *Comment*. Consumers, in turn, have two attribute elements: *ProducerAddress* and *Comment*. There is a one-way dependency between those two components: if a consumer exists then there must be a producer with the same address, otherwise the configuration is not valid. In addition, the configuration model defines the allowed relationships (and multiplicities) between the element types.

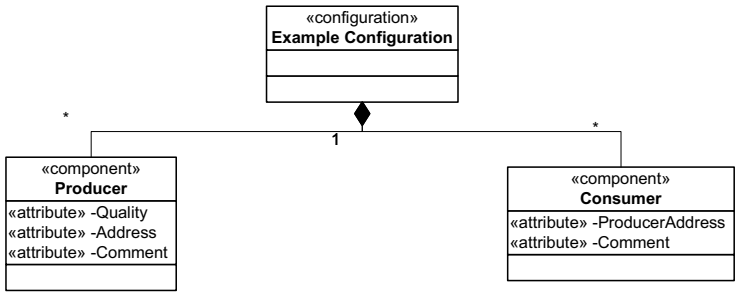


Fig. 2. Example configuration model

2.3 Configuration Pattern Specification and XML Schema

A configuration model is used to generate an XML schema and a configuration pattern specification for a product-line. In our approach, XML schema is used to validate existing XML configuration descriptions whereas configuration pattern specification is used to generate new valid configuration descriptions.

Configuration patterns follow a pattern metamodel shown in Figure 3. We exploit here the generic pattern concept of [7]. The basic building block of a pattern is called a *role*. Roles are bound to artifacts in a specific domain (i.e. XML and UML artifacts). A role has a type, which defines what kind of artifact elements can be bound to the role. When applied to XML, the type of a role ensures that the role gets bound to specific XML elements only.

The multiplicity of a role gives the lower and upper limits for the number of instances of the role. In this work, we assume that multiplicity is defined with symbols '1' for exactly one, '?' for zero or one, '*' for zero or more and '+' for one or more. If nothing is specified a role has multiplicity value of '1'.

Roles are organized hierarchically; each role may have a number of child roles. The hierarchical organization of roles must follow that of artifact elements; if role *R* is bound to an artifact element *E*, then the child roles of *R* can only be bound to the child elements of *E*.

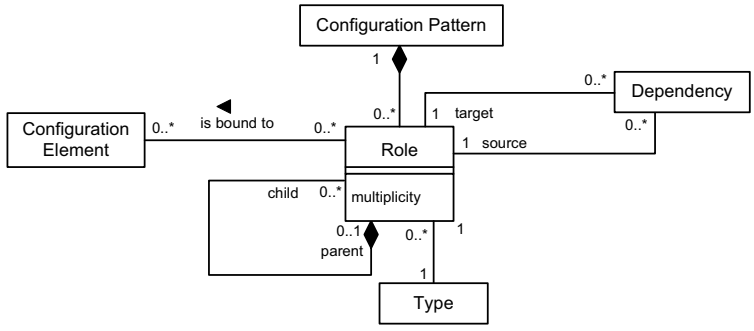


Fig. 3. Conceptual model for configuration pattern

The roles in a configuration pattern may have dependencies, which represent dependencies between elements in a configuration. The pattern tool generates a task for each role that can be bound in the current situation, taking into account the dependencies between the roles. For example, if an attribute in component A needs another attribute in component B, the creation of A affects tasks for creating B and its attribute are created.

Here we apply patterns for describing possible configuration elements and their multiplicities and relationships in a configuration instance. Thus, a configuration pattern contains the information - extracted from the configuration model – that is needed to guide the product developer to specify a legal product configuration. A configuration pattern can be viewed as a different representation of a configuration model, intended for a task-driven configuration tool.

A configuration pattern consists of a set of roles, which will be bound to actual configuration elements when the pattern is instantiated for a particular product. A role in a configuration pattern is either a component role or an attribute role, bound either to a component or to an attribute, respectively. An example configuration pattern for the configuration model discussed earlier is depicted in Figure 4. Initially, the pattern prompts to create a consumer description, after that tasks will be shown for the creation of its attributes and consumers. Various kinds of additional information

facilitating the interaction can be associated with a pattern, like prompt text, descriptions of the elements, instructions for performing a task (including links to helpful documents), default elements to be generated etc.

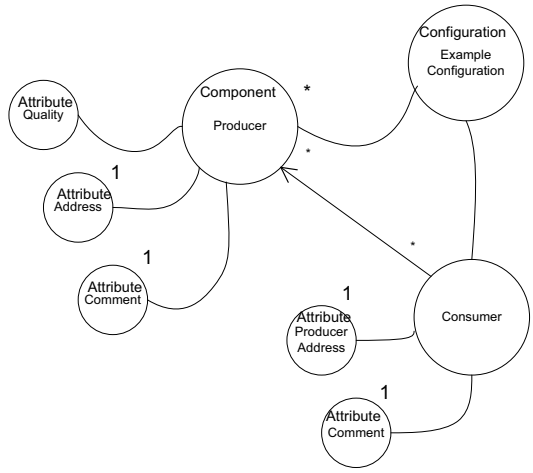


Fig. 4. Configuration pattern specification for example configuration

Configuration models are used to generate an XML configuration schema for the target product-line. In this case, the schema contains information about the components, attributes, their dependencies and multiplicities. A fragment of the XML schema for our configuration model is depicted in Figure 5.

```
<xsd:complexType name="ProducersAndConsumersType">

<xsd:element name="Producer" type="ProducerType"
minOccurs="0" maxOccurs="unbound" base="true"/>
<xsd:element name="Consumer" type="ConsumerType"
minOccurs="0" maxOccurs="unbound" base="true"/>
</xsd:complexType>

<xsd:complexType name="ProducerType">
  <xsd:element name="ProducingPower" type="xsd:string"/>
  <xsd:element name="Quality" type="xsd:string"/>
  <xsd:element name="Address" type="xsd:string"/>
  <xsd:element name="Comment" type="xsd:string"/>
</xsd:complexType>
```

Fig. 5. A fragment of example schema.

2.4 Configuration Creation Process

In our approach, two different kinds of tools are used: a modeling tool and a configuration tool. The modeling tool is used to create a configuration model, which

is an actual starting point for the configuration process. Then the configuration model is used to generate a configuration pattern specification and XML schema. The configuration tool is used to generate the XML configuration description, this generation process is guided by the configuration pattern specification.

An instance of the configuration pattern specification, is created for each configuration. The user generates new configuration components to the configuration by performing tasks that correspond to binding roles in the pattern specification. The user follows the task list and need not know anything about the configuration domain or XML schema.

As a configuration instance is being created, new XML fragments are added to the configuration file. Since manual modifications to the configuration file are possible, the XML schema is used to ensure that the manual additions did not violate the rules of the product-line.

3 VARMA Configuration Tool

In order to demonstrate our approach, we have implemented a task-driven configuration tool called VARMA [20]. The tool has been implemented as an extension of an existing environment known as JavaFrames [12], a pattern-based development environment originally intended for guiding framework specification. VARMA is built on top of Eclipse [5] and uses GEF [9] as an editor framework. The task engine of JavaFrames is used to generate tasks in the tool.

The VARMA tool consists of three Eclipse plugins. The Editorplugin component provides an editor, which represent a graphical visualization of the configuration. The Roleplugin component implements role types used in the configuration pattern. It also handles all functionality related to tasks and patterns. The last plugin, VARMA XML Editor is a text editor, which implements features for handling XML, for example, folding of XML blocks. The overall architecture of the VARMA tool is shown in Figure 6. A screen shot of the tool is presented in the next section.

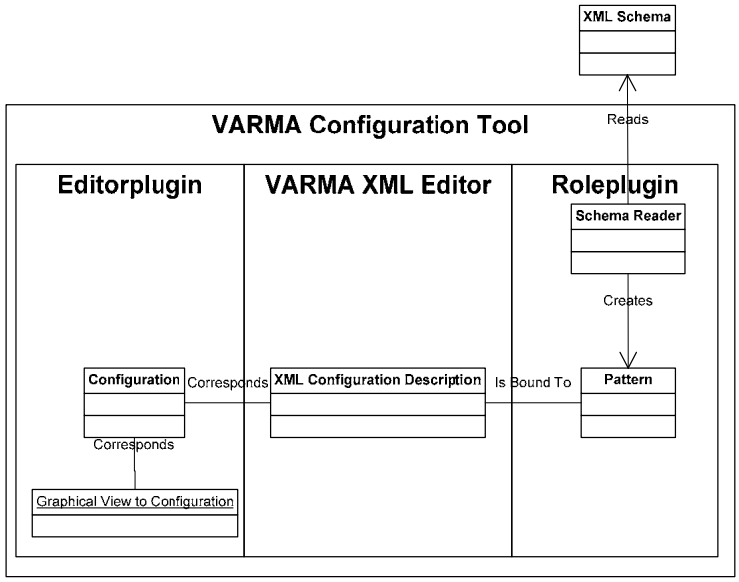


Fig. 6. Architecture of VARMA configuration tool

Producing a configuration for a product in the product-line is done by binding roles in the configuration pattern. By performing a task the user generates the default element to be bound to the role. The configuration elements specified in the configuration model are always bound to the corresponding roles, no such element can exist without a role counterpart in the current implementation. If the user desires to remove certain parts of the configuration, she has to simply delete the corresponding role binding using the VARMA tool. The configuration instance, the XML configuration description, and the pattern instance are thus always synchronized. The VARMA tool checks automatically the validity of the created configuration. When no mandatory tasks are displayed, the configuration conforms to the rules of the product-line.

As mentioned earlier, we propose generation of the configuration pattern specification and the XML schema of configuration from configuration model. However, our implementation is partial in that it lacks the use of a configuration model. At the moment a system architect, who is an expert in the configuration domain, writes the XML schema for a configuration description. After this a tool is used to generate the configuration pattern specification.

The information given in an XML schema specifies whether a component is a top-level (or root) component or not. In the example configuration model *Producer* and *Consumer* are root components and SubProductA is not. Also dependency information is given in the XML schema, and possible additional informal information documenting the components (to be displayed by the configuration tool) can be added there, too.

The configuration pattern specification is created from the XML schema using the following algorithm. Its idea is to walk through the whole XML schema and create a corresponding role for each XML element.

Pattern Specification Generation

```
  /* this pattern is the configuration root and it will
  contain all created roles. */
  ConfigurationPatternRoot confRoot

  ReadSchema(Document document)
    Element elementRoot = document.getDocumentElement()
    confRoot = createRootPattern()
    NodeList nl = elementRoot.getChildNodes()
  /* create childpatterns for base components and those
children */
    Foreach Node n in nl
        If (n instanceof basecomponent)
          models.createChildPattern(confRoot, childn)
          FindSubComponents(childn)

  /* method for creating roles for subcomponents. */
  FindSubComponents(Node n)
    NodeList nl = n.getAllChildren()
      Foreach Node childn in nl
        models.createChildPattern(n, childn)
        /* creation of tasks for attributes if needed. */
```

4 Case study: John Deere Harvester Condition Monitoring Application Product-Line

VARMA was evaluated with an industrial software product-line built for John Deere's harvester condition monitoring applications. Each application in this product-line is configured using a complex XML description, consisting of dozens of measurement descriptions and other XML elements. The configuration elements also have many hidden dependencies, which are only known by experts.

The condition monitoring system configuration model consists of six base components and several dozen child components. Each component contains several attributes. The XML schema of this configuration is over two hundred lines long and thereby configuration description can be very large, being thus an excellent case for our approach. A fragment of the configuration model of the condition monitoring system is shown in figure 7.

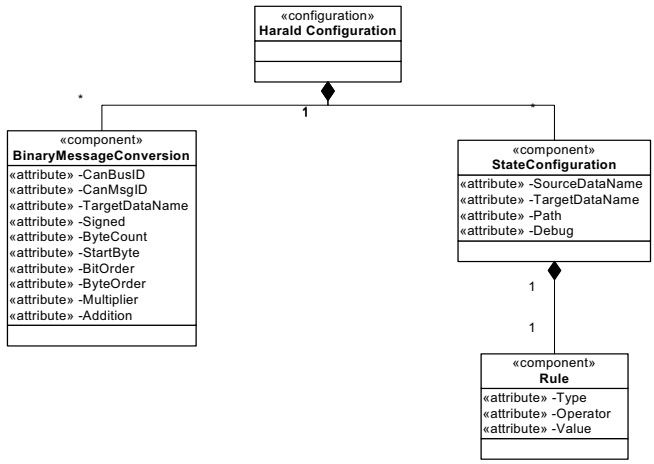


Fig. 7. A small fragment of condition monitoring system configuration model.

Configurations in this platform may consist of one file or they can be distributed in several files using regular include mechanism. The usage of several of files has helped the user to divide huge configurations to more controllable pieces. Because VARMA assists the user in configuration controlling, dividing the configuration to several files is not needed any more. Figure 8 shows the user interface of VARMA. The user has created a small condition monitoring configuration consisting of several components and attributes.

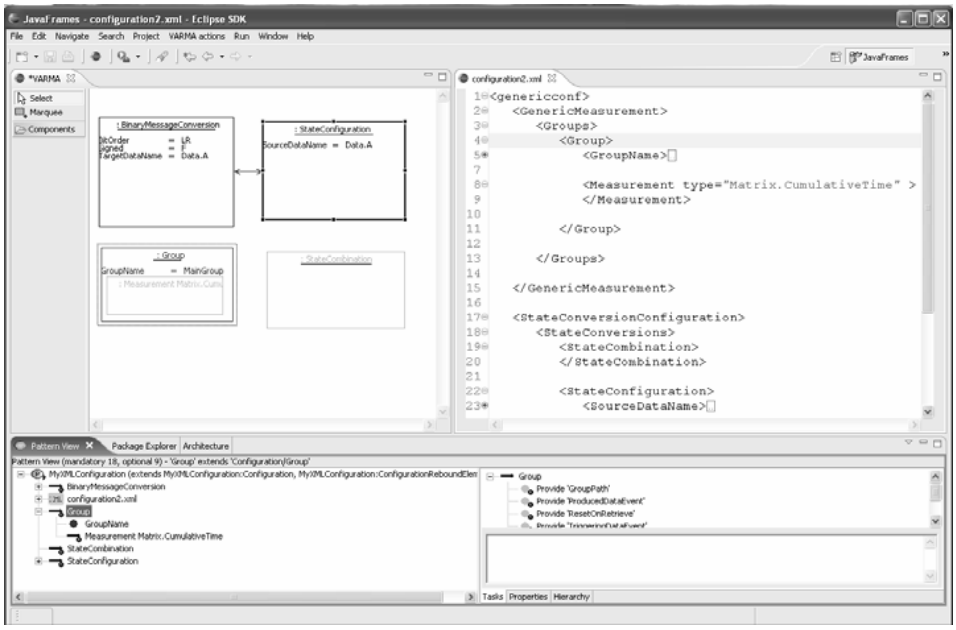


Fig. 8. VARMA user interface

In this configuration there is a dependency between *BinaryMessageConversion* and *StateConfiguration* components. *BinaryMessageConversion's* attribute *TargetDataName* and *StateConfiguration's* attribute *SourceDataName* must have the same value. In the diagram the dependency is represented as an arrow. This dependency is bidirectional; if one of those components with the attribute exists tasks for creating the other one and its corresponding attribute appear. Because dependency is bidirectional it has arrows at both ends. Schema configuration and VARMA tool enable also unidirectional dependencies. The visual representation of a configuration in VARMA is a read-only view to an instance of the configuration model. It is easier for the user to see contents of a huge configuration by looking at a graphical representation rather than a long XML file.

XML configuration description is created step-by-step by performing tasks. Typically the user action for performing a task is a single click and possibly inserting a value for the created element. We studied the required number of tasks as a function of the length of the produced XML description, to evaluate the feasibility of the task-driven approach. As shown in Figure 9, for a typical configuration description in the case study platform, the number of tasks in the creation process increases actually slightly slower than the length of the XML configuration description. Given that performing tasks is normally much easier and faster than writing the corresponding XML elements, even with proper XML tool support, this suggests that the task-driven approach yields clear benefits. On the other hand, when compared to a pure generative approach where a configuration description is generated directly from a high-level description (say, from an instance of the configuration model), the value of the task-driven approach depends on the importance of having the XML description as an editable, visible software artifact in the development process. Note that even in the case of a purely generative approach, the construction of the high-level description could be supported by a task-driven tool.

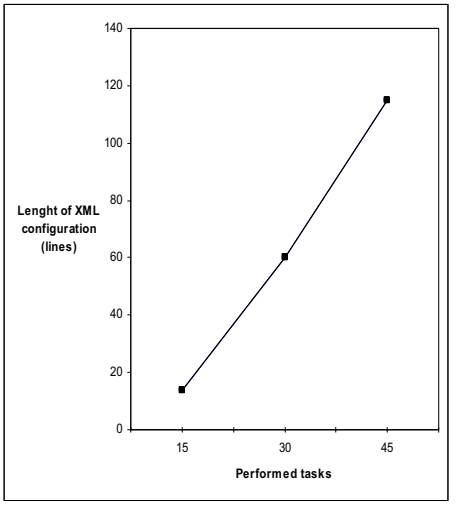


Fig. 9. Number of performed tasks

5 Related work

XML descriptions have been widely used to express variability information and to manage configurations in reusable software systems. Zhang and Jarzabek [23] present an XML-based Variant Configuration Language (XVCL) for handling variants in software product families. In XVCL, so-called x-frames are used to accommodate commonality and variability in a domain. This is similar to the kind of information represented in our XML configuration files. In [3] Choi et al. present an XML-based configuration management system for distributed systems known as XCONF (Xml-based CONFiguration management system). Their approach uses the Simple Object Access Protocol (SOAP) as a communication method to manage the complex relations of a subsystem with the information of other subsystems in order to support automatic system reconfiguration. This resembles our problem of managing hidden dependencies among configuration components. Another XML-based variation management approach has been used to manage middleware component configuration in interactive digital television systems [1]. Similarly to our work, an XML schema has been used to provide a means for defining the component model specification. Compared to these approaches, the VARMA tool offers different kinds of views (graphical view, tasking view, and binding view) to help developers manage the complexity of component configuration instead of working with low-level XML descriptions.

Visualizing the low level information in XML documents has been the subject of many research works. In [14] and [6], for example, a graphical representation using tree-maps is used to visualize and query XML structures. For similar purposes, a number of commercial XML tools have been developed [21]. Tools such as oXygen XML [19] and XMLSpy [22] are able to visualize, model, edit, transform, query, and debug XML documents. Compared to these general purpose XML tools, the tool capabilities of the VARMA have been developed to support the purpose of guided XML-based variation management. The graphical view of VARMA, for instance, is in a read-only mode and cannot be used to manipulate the XML data. Also, the generative and the tracing capabilities of VARMA suits better the kind of challenges faced in XML-based configuration management.

The model-driven approach to configuration management presented in this paper has been carried out in the domain of forest harvesting. Built on top of a general component configuration profile, our configuration model can be regarded as a domain specific language (DSL) [11] for the condition monitoring system. The purpose of our DSL is to manage the complexity of condition monitoring system configurations by generating XML descriptions (using configuration patterns) and validating existing configurations (using XML schema). Indeed, existing meta-case tools such as MetaEdit+ [17] and GME [10] can be used to solve our XML-based configuration problem. These tools offer a framework for defining concepts, properties, and rules of well-defined domains. Compared to these advanced multi-purpose tools, our approach aimed at a more lightweight tasking mechanism suitable for the problem of XML-based configuration management. In contrast to these tools, our approach supports interactive configuration description, rather than one-shot generation. We argue that interactive approach is more suitable especially for novice

users, because the user is expected to perform small tasks instructed by the tool, rather than to give complex specification.

Note that our three-level modeling approach provides a basis for a family of configuration languages, rather than a single language. Thus, new configuration languages can be easily developed for new platforms, if needed.

Using configuration patterns, the developer builds an application by gluing together existing platform components. This kind of black-box development technique is widely practiced in the area of object-oriented frameworks, which are a popular implementation mechanism for capturing domain variability [8]. Framework specialization is the process of adapting a framework to meet the requirements of a specific application (i.e. decisions regarding variation points). In our example, the decision on the variation points concerns setting the values of certain properties of existing components and defining the way these components are connected. Various kinds of tool support for framework specialization have been proposed. In [15], for example, Krasner and Pope proposed a cookbook approach. A cookbook typically contains information descriptions and guidance on framework specialization. The different functionalities supported by a framework can be implemented in the specialized application by following different recipes. A recipe in a cookbook approach is comparable with the support for user interactions during the generative use configuration patterns. In [18] Ortigosa and Campo introduce another approach providing guidance for framework specialization called SmartBooks, which are a more dynamic approach to cookbooks. These tools, typically work on code level. Our purpose, instead, is to generate XML descriptions for component interaction, in addition to a high-level graphical representation of the XML descriptions. In addition to generative capabilities, we also use XML schema for validation purposes. In contrast to these tools our approach exploits metamodel-based specifications.

6 Conclusions

Managing configurations in software platforms, especially when working at a low level of abstraction such as XML descriptions, is considered as a laborious and an error prone process. In this paper, we have presented a model-driven interactive approach to the creation and validation of XML-based configuration descriptions in software product-lines, aiming at more user friendly and controlled product derivation process. For this purpose, we have defined a configuration process that allows both generation and validation of configurations. In order to evaluate our approach, we have developed the VARMA configuration management tool. Even with the partial implementation of the proposed configuration process, the first experiment of the tool showed promising results.

As future work, we are planning to develop further the proposed configuration process and tool environment. This will be primarily guided by the kind of feedback we get from concrete usage of the tool environment.

Acknowledgments. The INARI project was funded by TEKES (Finnish Funding Agency for Technology and Innovation) and several companies: Nokia Reserch Center, Nokia Networks, John Deere, TietoEnator and Plenware.

References

[1] Borelli, F. Lopes, A. Elias, G. An XML-based component specification model for an adaptive middleware of interactive digital television systems. In *Proceedings. of AINA 2004*. p. 457- 462. March 2004. Fukuoka, Japan. IEEE Computer Society.

[2] J. Bosch. "Design & Use of Software Architectures: Adopting and Evolving a product-Line Approach". Addison-Wesley, 2000.

[3] Choi, H.M., Choi, M.J. and Hong, J.W. Design and implementation of XML-based configuration management system for distributed systems. In *Proceedings of NOMS 2004*. p. 831-844. April 2004. Seoul, Korea. IEEE Computer Society.

[4] Clements P., Northrop L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley 2002.

[5] Eclipse WWW site.,http://www.eclipse.org/.

[6] Erwig, M. A Visual Language for XML. In *Proceedings of VL 2000*, p. 47-54. September 2000. Seattle, Washington, USA. IEEE Computer Society.

[7] Hammouda, I., Hautamäki, H., Pussinen, M., Koskimies, K., Managing Variability Using Heterogeneous Feature Variation Patterns, M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 145–159, 2005.

[8] Fayad, M., Schmidt, D. and Johnson, R., editors. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, New York, 1999.

[9] Eclipse GEF www site, http://www.eclipse.org/gef/.

[10]GME: The Generic Modeling Environment, Institute for Software Integrated Systems WWW site. http://www.isis.vanderbilt.edu/Projects/gme/, 2007.

[11] Greenfield, J., Short, K., Cook, S.K., Crupi, J.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004.

[12] Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A.,Viljamaa J.: *Generating application development environments for Java frameworks.* Proc. GCSE 2001 (3rd International Conference on Generative and Component-Based Software Engineering), September 2001, Erfurt. Lecture Notes in Computer Science 2186, Springer, p. 163-176.

[13] INARI tool environment WWW site, http://practise.cs.tut.fi.

[14] Jelinek, J. - Slavik, P.: XML Visualization Using Tree Rewriting. In *Proceedings of SCCG 2004*. p. 65-72. Budmerice, Slovakia. ACM SIGGRAPH

[15] Krasner, G. and Pope, S. A cookbook for using the model-view-controller user interface paradigm in samlltalk-80. *Journal of Object-Oriented Programming*, pages 26–49, August/September 1988.

[16] Korhonen, M., Harald Framework, Nov 2005.

[17] MetaEdit+. MetaCase Consulting, MetaCase Consulting WWW site, http://www.metacase.com, 2007.

[18] Ortigosa, A. and Campo, M. A step beyond active-cookbooks to aid in framework instantiation. In *Proceedings of TOOLS Europe 1999*, p. 131-140. June 1999. Nancy, France. IEEE Computer Society.

[19] oXygen XML editor WWW site. At http://www.oxygenxml.com/. 2007.

[20] Vanhatupa, J-M., VARMA: Pattern-driven tool support for XML-based variation management, MSc Thesis, April 2007.

[21] W3C. XML tools WWW site. At http://www.w3.org/XML/Schema#Tools. 2007.

[22] XMLSpy XML editor WWW site. At http://www.altova.com/products/xmlspy/xml_editor.html. 2007.

[23] Zhang, H. and Jarzabek, S. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming.* 53(3): 381 – 407, 2004. Elsevier Science.

Versions and differences of models through maintenance and evolution

Harald Störrle, Tarja Systä, Sven Wenzel

Special Session on Model Comparison at the
Nordic Workshop on Model Driven Engineering 2007 (NW-MODE'07)

In 1981, James Martin, then a famous and influential IT consultant, entitled one of his many books *Application Development without Prgrogrammers*. This vision of replacing—or at least supplementing—text-based programming by diagram-based modeling has been with us to this day, periodically being revitalised by new acronyms, the latest of which are Model Driven Architecture (MDA), and Service Oriented Architecture (SOA).

While the progress towards real-life industrial-scale heavy-duty software modeling can not be overlooked, there are still significant obstacles to make this vision of old become reality. Two of these obstacles are the unsolved issues model comparison and version control.

Techniques to support comparison and version control of textual documents have been widely used in e. g. text editors and configuration management systems. Corresponding techniques and tools are also needed for (software engineering) models. Since models and diagrams significantly differ from textual documents, the techniques used for document comparison are not directly applicable to models.

Techniques and tools for comparing and differencing models are used for various software engineering purposes, including version management, model refinement, software analysis, and reverse engineering. It is unclear, however, how the various approaches differ, and what the specific benefits of each of them is. It is not even clear, what the range of applications is, to which these tasks may fruitfully be applied.

It seems, that these questions are largely *empirical* questions, so that the answer should be determined by field surveys, case studies, and, possibly, controlled experiments. This immediately raises the question for benchmark models: what criteria and qualities constitute a "good" benchmark? Which kind of empirical research method is likely to be best suited to contribute to these questions?

We have invited four experts to discuss these questions and all related issues at this special session at NW-MODE'07. Our panelists are

- **Claudio Riva** Principal Scientist, Nokia Research Center
- **Petri Selonen** Assistant professor, Tampere University of Technology
- **Harald Störrle** Professor of Software Engineering at the Innsbruck University and senior consultant at mgm technology partners GmbH in Munich
- **Eleni Stroulia** Associate Professor, University of Alberta

For further information please consult the special session web page at `http://www.ituniv.se/~miroslaw/node.htm`.

Special session on

Quality in Modeling

Organized by:
Ludwik Kuzniarz, Blekinge Institute of Technology, Sweden
Parastoo Mohagheghi, SINTEF, Norway
Miroslaw Staron, IT University of Göteborg, Sweden

Introduction
Quality management in model-driven projects needs to be refocused to reflect the facts that models are the primary artifacts and some of the models are not executable or directly for code generation (e.g. analysis models). The understanding of the notion of quality and its working definition from the IEEE and ISO standards needs to be fine-tuned to reflect the quality of different kinds of models – analysis, design, platform, meta-models, and implementation models. The adjustments also need to be done to the way in which quality is managed in software projects - in particular how it is measured. In order to ensure proper or satisfactory level of quality a number of trade-offs can occur. Identification of the potential trade-offs and their possible addressing and handling is important from the developers and industrial perspectives.

Purpose
The purpose of this special session is to discuss the issues related to the following:
1. Understanding of quality in the context of model-driven software projects
2. Assessing and measuring quality
3. Trades-offs related to quality

The expected outcome is a set of indicated trade-offs together with related win-loose factors when working with quality in model driven projects.

Organization
The session will be organized as a guided panel discussion. It will start with the position statements of the panellists followed by a discussion based on the questions from the audience.

The leading topic will be "Trades-offs related to Quality in Modeling" and the panelists will be asked to comment on the following:
- What do you mean by a good model?
- What are / could be trade-offs in handling / ensuring model quality?
- What are / could be the costs related the trade-offs?