Dipl.-Ing. Michael Thonhauser

A Runtime System for Model-based Software Components in Mobile Environments

Dissertation

vorgelegt an der

Technischen Universität Graz

zur Erlangung des akademischen Grades
„Doktor der Technischen Wissenschaften“
(Dr. techn.)

durchgeführt am Institut für Technische Informatik
Vorstand: O. Univ.-Prof. Dr. techn. Reinhold Weiß

Dezember 2010

Abstract

Model Based Software Engineering (MBSE) approaches apply different techniques on a model of the developed application. This model contains an abstract description of the software specification and therefore fosters reuse of the described software artifacts on multiple platforms.

For the platform specific development and deployment of such applications Component Based Software Engineering (CBSE) approaches are available, each supporting the assembling of different specified components to an application running on a specified target platform. Because of the trend of ubiquitous computing, which enables the creation of new distributed business processes, the number of target platforms is increasing. As these business processes are executed with the help of different mobile devices, software support for these business processes requires engineering approaches to specify an application in a platform independent way while supporting the deployment and dynamic configuration of these application artifacts.

The approach presented in this work introduces a model-based middleware architecture, relying on a platform independent definition of the modeled artifacts. By applying event based communication mechanisms and model interpreting techniques the platform independent specification is transformed into a platform specific representation at runtime. The explicit definition of events and data for each artifact in a model fosters the reuse of the specified software artifacts while the meta-model enables the portability of the interpreting code between different platforms.

The proposed approach has been evaluated in the mobile part of the software WAMAS built by the company Salomon Automation, a software vendor in the application domain of logistics, and in the extension of an existing RFID middleware maintained by the company RF-iT Solutions.

Kurzfassung

Modellbasierende Software Entwicklung (MBSE) verwendet verschiedene Verfahren, um ein Modell einer Applikation auszuwerten. Dieses Modell enthält die Spezifikation der Software auf einer abstrakten Ebene und erleichtert damit die Wiederverwendung der definierten Applikationsteile auf mehreren Plattformen.

Für die plattformspezifische Entwicklung und die Verteilung dieser Applikationen werden komponentenbasierende Softwareentwicklungstechniken (CBSE) verwendet, wobei jeder Ansatz die Zusammenschaltung verschiedener Softwareteile zu einer Anwendung auf der Zielpattform unterstützt. Die Anzahl unterschiedlicher Plattformen für mobile und eingebettete Systeme nimmt durch die wachsende Bedeutung von "Ubiquitous Computing" stark zu, wobei dieser Trend auch neue Organisationsabläufe ermöglicht. Um die Entwicklung von Applikationen für diese Organisationsabläufe bestmöglich unterstützen zu können, ist eine plattformunabhängige Spezifikation der Applikationsteile notwendig, wobei aber trotzdem die Verteilung und dynamische Rekonfiguration der Komponenten unterstützt werden soll.

Der in dieser Arbeit vorgestellte Ansatz verwendet eine modell-basierte Middleware Architektur, die auf die plattformunabhängige Spezifikation der Applikationsteile in Form von Modellen aufsetzt. Durch Anwendung event-basierender Kommunikation und durch Verwendung modellinterpretierender Techniken wird diese plattformunabhängige Beschreibung zur Laufzeit in eine plattformspezifische Repräsentation übersetzt. Die explizite Definition von Events und Daten in eigenen Modellen ermöglicht dabei eine exaktere Wiederverwendung der entwickelten Softwareteile, während das Metamodell den Umfang der plattformspezifischen Codeteile (die im Allgemeinen aus dem Code des Interpreters bestehen) einschränkt.

Der vorgeschlagene Ansatz wurde in der mobilen Applikation des Lagerverwaltungssystems WAMAS der Firma Salomon Automation und in der RFID Middleware Lösung YouR OPEN der Firma RF-iT Solutions evaluiert.

Extended Abstract

Motivation

In cooperation with *Salomon Automation GmbH*, a logistics software vendor for warehouse management systems, and *RF-iT Solutions*, a provider of a middleware for RFID-based systems, a new architecture enabling better maintenance and deployment of software components on various mobile and embedded devices should be established. These mobile devices are Personal Digital Assistants (PDA) used by the workers in a warehouse, RFID readers installed at different partners of a business process, wireless sensor nodes established in the supply chain or mobile phones used by the suppliers of a warehouse operator.

According to previous work (data) modeling is a useful basis for creating a software product family. These software products are applied in desktop and server computing environments and are built upon a feature enabled data model. Each model is contained in software components, which are also providing the definition of the user interface and the behavior in the form of platform specific code artifacts. As a consequence the porting and migration of these components is impeded by the assumption of a static component platform, which may not be supported by all members of a distributed business process.

To fill this gap, an architecture of a runtime system for model-based software components is proposed in this work. This architecture can be applied to execute applications built of components, which contain a formal specification by models of their used data, user interface and behavior. Because the configuration of the runtime system members is also based on the meta-model used by these software components, a migration of software components at application runtime between different platforms is supported by this approach.

Related Work

Model Driven Software Development (MDSD) is an industrially accepted approach for software engineering using formalized methods. The majority of MDSD approaches follows the Model Driven Architecture (MDA) concept. In this concept the first activity is the specification of a platform independent model, which is transformed to a platform specific model by applying several generators. The layered Meta Object Facility (MOF) approach is used for creating the models. This approach is also used the basis for the Unified Modeling Language.

While MDSD facilitates models for the abstract specification of system architectures, their platform specific artifacts are often realized by applying Component Based Software Engineering (CBSE) techniques. The basic element in these approaches is a software component, which is a unit of execution with well defined interfaces. Usage of software components is driven by the requirements of improving reusability of developed software artifacts. Software components are combined with the help of assembly descriptions. They are specified in the development phase and are resolved in the deployment phase of a CBSE

process.

Mobile business applications are typically realized in a distributed system architecture. Service Oriented Architectures (SOA) and Grid Computing have been proposed for the construction of such distributed systems. SOAs are dealing with business processes distributed over existing and new heterogeneous systems, which are under the control of different owners. They are build on services providing well defined interfaces. While this aspect is also targeted in CBSE, the proposed loose coupling of SOA services is contradictory to the assemblies used in CBSE. Grid Computing has evolved for the distribution of scientific computational tasks, but is also used for distributed data access in heterogeneous network. A Virtual Organisation is formed by the members of a Grid.

A runtime system for model-based software components in mobile environments

Mobile business applications are applied in a distributed business process, with several participants realizing different functionalities in a multi-layered (or multi-tiered) architecture. The development and deployment of components used in the realization of such distributed processes is targeted by this work.

A model-based approach for the distributed component runtime environment is introduced. This approach enables the specification of the components by domain experts, who are not required to be professional software developers. Model-based techniques are used in this specification to avoid the usage of platform dependent code. As a consequence the effort required for deploying these components on another platform is reduced to the porting of the runtime environment elements and dedicated device access methods. This task is handled separately by specific platform developers.

The architecture of the proposed system provides the following methodologies to enable a distributed runtime environment for software components in mobile environments:

- *Specification of components by multiple models* [1] [2]

  The functional specification of components realized by this approach is defined with distinct models for various aspects of the functionality. Each modelled aspect is interpreted at runtime, with the model interpreter being based on the meta-model of this model. All model interpreters of a specific component are executed by a runtime node participating in a distributed runtime environment.

- *Transient extension of (meta)models* [3] [4]

  Participants of a runtime environment are configured by a meta-model containing the elements of the models, which are used for the functional specification of the components. The mechanism of Transient Model Extension has been proposed for temporal addition of elements to a meta-model. This methodology is used in the

[1] Interpreting Model-Based Components for Information Systems. *ECBS 2009, San Francisco, California, USA, April, 2009.*

[2] Towards a Generic Model Interpretation Runtime Architecture. *Work in Progress Euromicro 2009, Patras, Greece , September, 2009.*

[3] Model-Based Data Processing with Transient Model Extension. *ECBS 2007, Tucson, Arizona, USA, March, 2007.*

[4] Implementing model-based data structures using transient model extensions. *Journal of Software, 2007.*

configuration of runtime participants and to enable the specification of additional runtime functionality by the component developer.

- *Component connectors based on model compatibility* [5]

  To enable a loose coupling of the executed components, the communication between the components is distributed by the runtime nodes. Because each component is based on its own model, the communicating components make use of the runtime node functionality to enable the communication with compatible components.

- *Specification of model views for distributed components* [6] [7]

  All runtime nodes are part of a distributed system and are members of a Virtual Organisation. Thus additional views on the component model can be specified by the developer, describing the visibility of model elements and quality of service attributes used by other component developers referring to the given component and by the component runtime nodes during execution of the components.

Implementation
A design of a runtime system for model-based software components is provided, which has been implemented following the architectural styles specified in the previous section. This runtime system is targeted for mobile and embedded devices. Sharing of provided device resources and the execution of third party components is eased by this approach. Tools for deploying the runtime and for the development of the components are described, and examples of their usage are discussed.

Case study
The proposed architecture has been applied by Salomon Automation GmbH for an industrial prototype of the mobile client of the WAMAS software architecture. Another application of the architecture has been provided for the middleware of RFIT - Solutions to extend the range of their supported devices by moving middleware functionality to the reader.

Both case studies outlined the benefit of using multiple models for the functional specification resulting in a reduction of platform specific code. The results also demonstrated the importance of good tooling support for creating the functional models and identified the performance of the model interpreters as a key aspect required for application of this approach in industrial projects.

[5] Data Model Driven Enterprise Service Bus Interceptors. *Euromicro 2008, Parma, Italy, September, 2008.*

[6] Model-based data access in mobile grid applications. *PDCN 2008, Innsbruck, Austria, February, 2008.*

[7] A Model-Based Architecture supporting Virtual Organizations in Pervasive Systems. *ICECCS 2010, Oxford, UK, March, 2010.*

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Acknowledgments

This thesis was written at the Institute for Technical Informatics, Graz University of Technology, in cooperation with the companies Salomon Automation GmbH and RFIT Solutions GmbH.

For his encouragement and scientific support I would like to thank my supervisor and head of the Institute for Technical Informatics Prof. Reinhold Weiß. For his continuous support in my research and valuable discussions on our research project I also express special thanks to Christian Kreiner, who has also perfectly managed the fundings of the various projects.

The support of my friends and colleagues Zsolt Kovacs, Martin Schmid, Gernot Schmölzer and Egon Teiniker has been very valuable in the preparation of the publications, while the master theses of Stefan Kremser and Andreas Leitner provided worth-while experiences. I also acknowledge the work of Manuel Menghin, Ulrich Krenn and Nicolas Pavlidis, who did a great job on the implementation of the Model-Based Component Container.

Additionally I would like to thank my colleagues at the Institute of Technical Informatics and at the company Salomon Automation GmbH for providing a pleasant working atmosphere and research environment. For his moralic support I express special thanks to Peter Pühringer.

For their initial financial support of this research thanks are expressed to Franz Salomon and Albert Burgstaller of the company Salomon Automation and especially to Matthias Weitlaner and Stefan Schnuderl of the company RFIT Solutions, who provided financial and technical support for the second case study. Finally I am extraordinary thankful to my wife Melanie, to my parents, my brothers and all other members of my family for their continuous support throughout the years of my studies.

Graz, December 2010 MICHAEL THONHAUSER

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A remarkable advance in the technology of mobile and embedded devices has been achieved in the last decade. Roy Want mentions in [Wan09] that the number of shipped smart phones in 2007 has been larger than the number of sold laptop computers, while stating that smart phones make about 10 percent of the cell phone shipments in the same year. In contrast to traditional cell phones, smart phones are equipped with an application processor enabling additional features like video recording or positioning technology.

Ebert and Jones present recent data about embedded software in [EJ09], stating that the volume of embedded software is increasing between 10 and 20 percent per year as a consequence of the increasing automation of devices and their application in real world scenarios. Ebert and Salecker give the following definition of an embedded system.

"Embedded systems are microcontroller-based systems built into technical equipment. They're designed for a dedicated purpose and usually don't allow different applications to be loaded and new peripherals to be connected. Communication with the outside world occurs via sensors and actuators; if applicable, embedded systems provide a human interface for dedicated actions."[ES09, p.14]

Although this definition implies that embedded systems are used as isolated units, there is also a trend to construct distributed pervasive systems by connecting several embedded devices as noted by Tanenbaum and van Steen [TvS06]. Also according to data presented by Ebert and Jones up to 70 electronic units are used in a car containing embedded software, which is mainly responsible for the value creation of the car and consists of more than 100 million lines of object code (requiring about 1Gbyte of storage).

Mobile and embedded devices in distributed pervasive systems support physical and logical mobility as discussed by Mühl et al. in [MFP06]. A physically mobile client (a.k.a terminal mobility) is supported by an infrastructure consisting of geographically distributed access points, which are interchangeably used while the client is moving, allowing the realization of location transparency. In contrast a logically mobile client is aware of the changing location enabling automatic adaptation to the new situation. Considering the ISO/IEC 42010 IEEE Std 1471-2000 definition of an environment [IEE07] as the definition of the scope of a system, used for the determination of the boundaries and the settings and circumstances of developmental, operational, political, and other influences upon that system, the following definition of a mobile environment can be considered.

Mobile environment: A mobile environment consists of devices and applications, which support the logical and physical mobility of their users.

1.1 Motivation

For further discussion the common usage of the layers architectural pattern [BMR+96] for specifying software architectures has to be considered. The following definition of software architecture is given in the ISO/IEC 42010 IEEE Std 1471-2000:

Software Architecture: "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.-[IEE07, p.3]

A three layered architecture, which is depicted on the left side of Fig. 1.6, following this definition will be used in this thesis. To provide a foundation of this layered architecture it is compared with other layered architectures discussed in literature. One architecture is proposed by Tanenbaum and van Steen for distributed systems, which are defined in the following way:

Distributed system: "A distributed system is a collection of independent computers that appears to its users as a single coherent system." [TvS06, p.2]

Another architecture is specified by Atkinson and Kühne while discussing the definition of platform in Model Driven Architectures (MDA) [AK05]. In their discussed architecture the **hardware layer** consists of hardware components (e.g. processor, memory, I/O devices), which are augmented with additional services (like threads, file systems or processes) by the **operating system layer**. A **virtual machine layer** can be used for isolation of the chosen operating system and hardware components. A language runtime system is provided by the **language support layer**, which also contains language constructs expressed as templates of low-level code. These constructs are used for the specification of libraries, frameworks and applications defined in the upper layers. Therefore required or optional additional predefined functions in the used programming language are contained in the **library layer**, but also middleware solutions are typically delivered as libraries providing several language constructs in the form of an application programming interface (API). In contrast to libraries elements in the **framework layer** contains active control code in a generic way to be used by a family of applications. Using this layered structure Atkinson and Kühne demonstrate, that each layer is defining a platform on its own. As a consequence a generic platform model is introduced by them, which is applied on every organizational layer leading to a full platform definition with the combination of these platform models. This generic platform model is summarized by Kleppe in her definition of a platform.

Platform: "A platform is the combination of a language specification, predefined types, predefined instances, and patterns, which are the additional concepts and rules needed to use the capabilities of the other three elements." [Kle08, p.69]

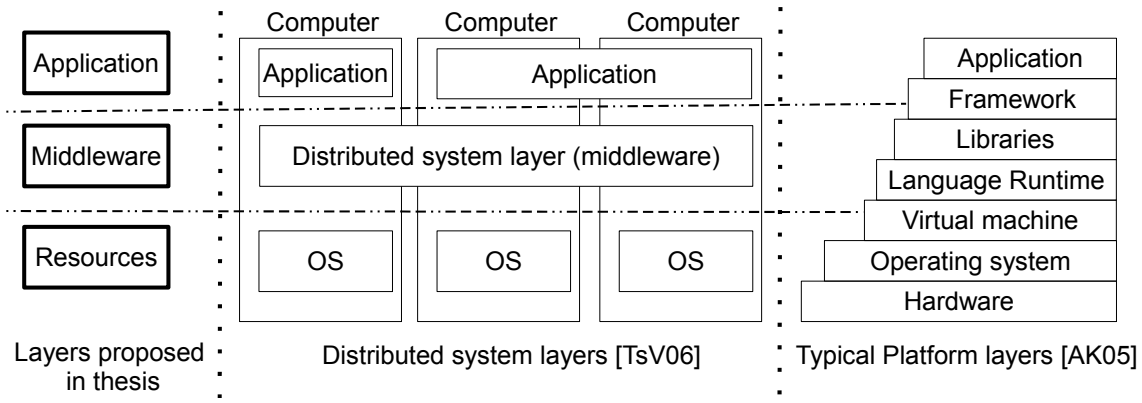


Figure 1.1: Comparison of runtime architectures

Fig. 1.1 provides a comparison of these two architectural definitions with the proposed layered architecture depicted on the left side demonstrating the feasibility of mapping the corresponding artifacts and responsibilities for each layer of the proposed architecture.

1.1.1 Reusability and evolution support through components

Platform evolution is characterized by changes on one or several platform layers of a device, requiring that these changes should not affect the role of this device in the distributed system. Several techniques have been proposed for supporting this evolution.

As the number of elements contained in the platform model of each platform layer for mobile and embedded devices needs to be optimized as a consequence of the resource constraints, usage of component based technologies targeting reuse and composition is an essential part of platform and application development for such devices. Additionally Liggesmeyer and Trapp have noted, that domain specific development of embedded software requires efforts, which should be payed off by the application of the developed framework or platform to related problem domains [LT09]. This requirement for reuse is solved by component based software development, allowing the composition of individual artifacts relying on well defined interfaces.

The following definition given by Szyperski in [Szy02] is often used as the basis for a discussion of CBSE aspects.

Software Component: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party".[Szy02, p.41]

Depending on the implementation for the execution environment of a chosen component approach, the notion of a component framework is distinguished from a component platform by Szyperski in [Szy02], with the former defined as a collection of rules and interfaces (contracts) governing the interaction of components. In contrast to component

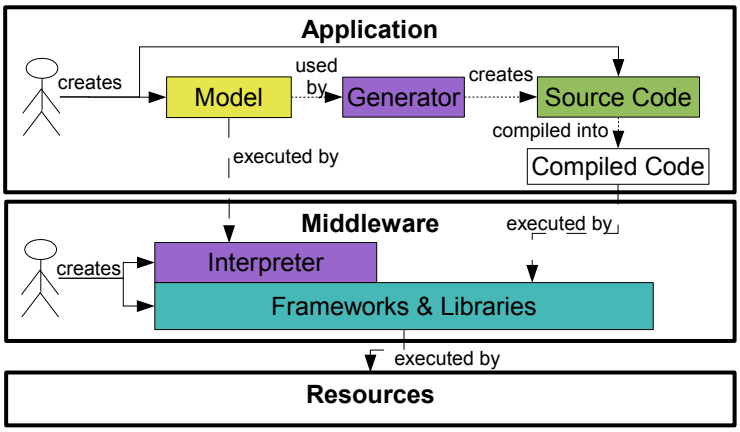


Figure 1.2: Involved layers in MDSD approaches

platforms, which are the foundations for components to be installed and executed on, component frameworks can also be organized hierarchically being components themselves. During platform evolution this fact gets also very important for enabling the adaptation of the component execution environment to changes in the lower platform layers (e.g. changed hardware or operating systems or additionally available programming language constructs). As an example of an evolution at the platform level, the usage of the hardware platform promoted by the Wear-Ur-World device (a.k.a. Sixth Sense) [MMC09] can be considered. This device has been developed at the Fluid Interfaces Lab of the Massachusetts Institute of Technology (MIT) and is bridging the real and the digital world by a system consisting of a mobile camera and a mobile beamer which are connected to a mobile phone running a software for recognizing hand gestures. Incoporation of this new platform in an existing architecture is eased by the layered approach of systems and by a component oriented design, which should allow to replace the typical graphical or textual user interface by a hand gesture aware implementation, minimizing the changes required to the application itself.

1.1.2 Domain specific development with models

As noted by Ebert and Salecker [ES09] embedded software is used for objectives requiring a long lifetime of the system, thus heterogeneity of the underlying platform has to be considered by the embedded software engineer. Also an ever-growing demand for new functionalities and technologies is noted by Liggesmeyer and Trapp [LT09], requiring efforts of raising the level of abstraction for the development of mobile and embedded software. For embedded systems this is accomplished by the usage of higher level programming languages like C or C++ instead of assembler, and also by the help of Model Driven Development (MDD) techniques.

In Model Driven Software Development (MDSD) models of the software are used as primary artifacts. There is no unique definition of a model, as reported by Muller et al. in [MFB09] while comparing several definitions of a model in the literature. But most defi-

nitions describe the use case of a model, which is applied to abstractly specify a distinct system. According to Muller et al. the process of modeling also aims to establish to represent something by something else. Dealing with the abstraction introduced by such models is provided by model transformators, which are applied at development time in the form of code generators or at runtime in the form of model interpreters as reported by Stahl and Völter in [SV06]. While the usage of code generators is a mature technology backed by the Model Driven Architecture (MDA) standard of the Object Management Group (OMG), the usage of models at system runtime has attracted interest in the research community in the last years as indicated by the following definition.

model@run.time: "A model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective."[BBF09, p.23]

Fig. 1.2 depicts the layers involved in these two approaches to MDSD. While generated code is usually targeted for the application layer using distinct libraries at the middleware layer, model interpretation is performed primarily by generic interpreters, which are also located in the middleware layer of the layered architecture proposed for this thesis.

In state-of-the-art MDSD approaches the definition of component interfaces is already based on specific software models, which are used for generation of the tooling support and platform specific skeletons in a general purpose programming language. In such CBSE approaches the implementation of the component containing the domain specific knowledge of a specific problem is mixed with programming constructs enforced by the used programming language. A good understanding of this language is required to successfully adapt the given code to changed requirements in the problem domain.

This problem is tackled by Domain Specific Languages (DSL), which are providing concepts and rules representing problems in the application domain instead of the underlying programming language. Kleppe has demonstrated in [Kle08] that DSLs can be defined by using models containing the elements of the abstract syntax and the concrete syntax of the DSL respectively. Because a specific view on a modeled system is defined by a DSL, combination of several DSLs (i.e. their corresponding abstract and concrete syntax models) can lead to a fully executable system as demonstrated by Hessellund in his proposal for Domain-specific Multimodeling [Hes09].

Models and especially DSLs are targeted for the usage by domain experts, who are working on a software project in one of several roles as defined by Fischer et al. in [FNY09]. The spectrum of possible roles is depicted in Fig. 1.3.

Beside pure software users and programming professionals different grades of expertise level in software development are seen, demanding mechanisms such as macros or Domain Specific Languages (DSLs) to support the user in customization of the software. The generic mechanism in the software required by this customization can be provided by the developed models serving as a basis for possible customization decisions.

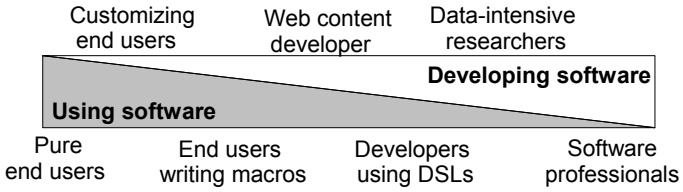


Figure 1.3: Spectrum of software related activities [FNY09]

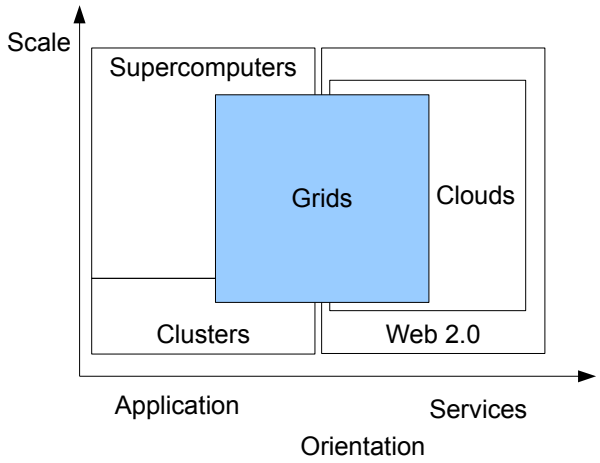


Figure 1.4: Comparison of distributed systems [FZRL08]

1.1.3 Connecting mobile and embedded systems

For the discussion of the support of applications using several mobile and embedded devices the categorization of distributed systems by Tanenbaum and van Steen in Distributed Computing Systems, Distributed Information Systems and Distributed Pervasive Systems has to be considered. Distributed Computing Systems are further categorized in Cluster Computing and Grid Computing systems, with the later being used for connecting heterogeneous computing systems. The following definition of a grid is provided by Foster and Kesselman:

Grid: "A distributed computing infrastructure that supports the creation and operation of virtual organizations by providing mechanisms for controlled, cross-organization resource sharing." [FK04, p.662]

The relation between Grids and other distributed computing paradigms is discussed by Foster et al. in [FZRL08] and is illustrated in Fig. 1.4. According to this figure Grids are covering both ranges regarding the scale and the orientation of the executed software.

The other areas in this figure are corresponding to the history of Grid systems as given by Kurdi et al. in [KLAR08]. The first generation of Grids was build upon supercomputers

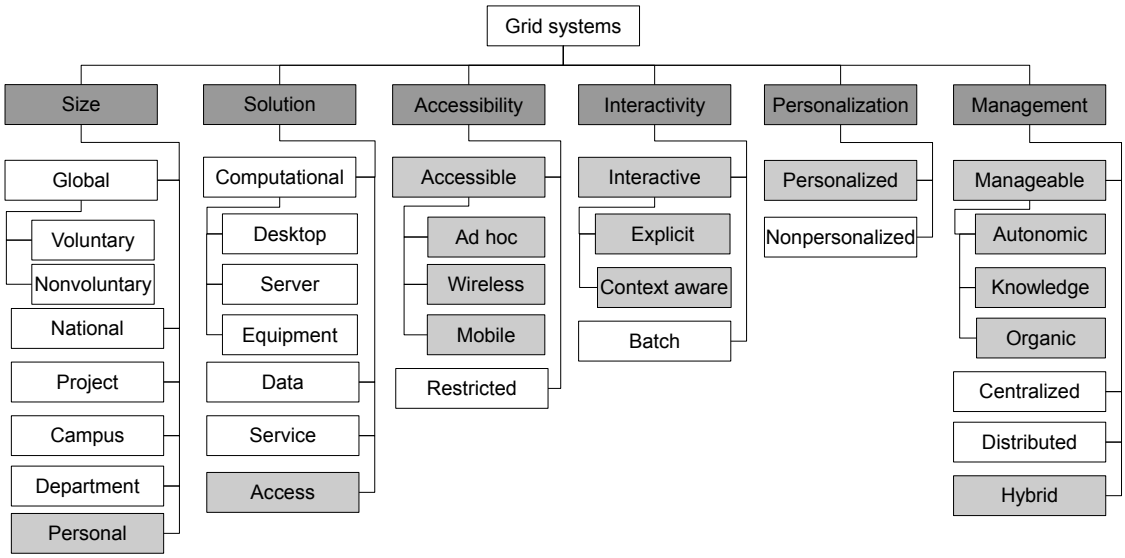


Figure 1.5: Classification of grid systems (classification criteria are dark grayed and emerging grid types and grid subtypes are light grayed) [KLAR08]

sharing their resources, which led to Computing Grids or Data Grids [MJR+04]. To enable the access to dedicated rare resources (like a hadron collider or a space telescope)[FK04] corresponding middleware functionality needed to be defined, which was the main task of the second Grid generation. A Grid middleware is build in a layered approach like the four layers proposed by Foster in [FZRL08]. The highest layer is named the **collective layer** and captures interactions across collections of resources and allows for monitoring and discovery of VO resources. The second layer named **resource layer** defines protocols for the publication, discovery, negotiation, monitoring, accounting and payment of sharing operations on individual resources. The third layer is named **connectivity layer** and defines core communication and authentication protocols for easy and secure network transactions, while the last layer known as **fabric layer** provides access to different resource types such as compute, storage and network resources.

Third Grid generation implementations rely on Web technologies to implement the different layers of the middleware (and are therefore related to the Web 2.0 paradigm as depicted in Fig. 1.4). According to a classification of grid systems presented by Kurdi et al. (depicted in Fig. 1.5) next generation grids (NGG) will be of personal size and are going to offer the access to distributed devices as the solution. NGGs will be accessible by any device using one of the according grid subtypes (ad hoc, wireless or mobile). They will also be interactive and personalized and should be manageable or use a hybrid approach of the different management approaches.

As a consequence of this evolution Grid systems will need to pay attention to the constraints of distributed systems made up of mobile and embedded devices. These constraints are noted by Tanenbaum and van Steen [TvS06] in their discussion of distributed pervasive systems formed by these devices. Such systems are indicated by unstable con-

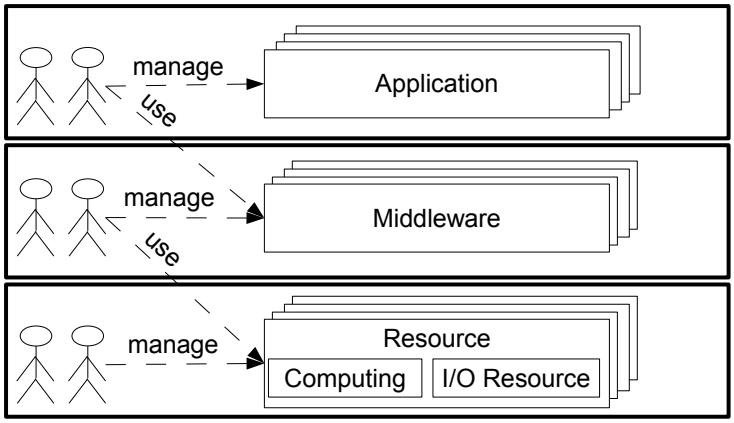


Figure 1.6: Architectural layers used in thesis

nections, resource constraints of the participating devices and the general lack of human administrative control (i.e. the configuration of such devices is allowed to device owners only).

1.1.4 Missing support for evolution in Virtual Organizations

Fig. 1.6 highlights two aspects of heterogeneity managed by the layered architecture proposed in Sect.1.1. One aspect is the number of artifacts contained in each layer, while the second aspect is made up by the people managing the artifacts at the same layer. The kind of management is defined by the state of the system. Management at development time is performed by designing and implementing the given artifact of a layer in software or hardware. At runtime the administration and usage of available instances of the defined artifacts by a (typically other) person has to be considered. Note that multiple persons are involved in the management operation of each layer as well as multiple artifacts. In a distributed system made up of mobile and embedded devices the involved people as well as the used artifacts are expected to be organizationally and geographically distributed, which are key features of virtual organizations as defined by Foster and Kesselman.

Virtual Organization: "A collaboration whose participants are both geographically and organizationally distributed." [FK04, p.672]

Bird e .al. note in their discussion, that "the original idea of a VO as a dynamic group of users with a common goal coming together for a specific, short-lived collaborative venture and then dissolving has never been realized owing to the complexity of deploying and authorizing such a dynamic structure."[BJK09, p.41] This statement is represented by the multiple persons and artifacts contained in each layer of Fig. 1.6. Each person and each artifact can be added, replaced or removed at system runtime, constituting a multidimensional evolution leading to the dynamic structure required by Bird et al..

This fact is also important in the business domain of logistics, where traditional business activities (e.g. handling of orders in a warehouse with paper-based forms) are replaced by

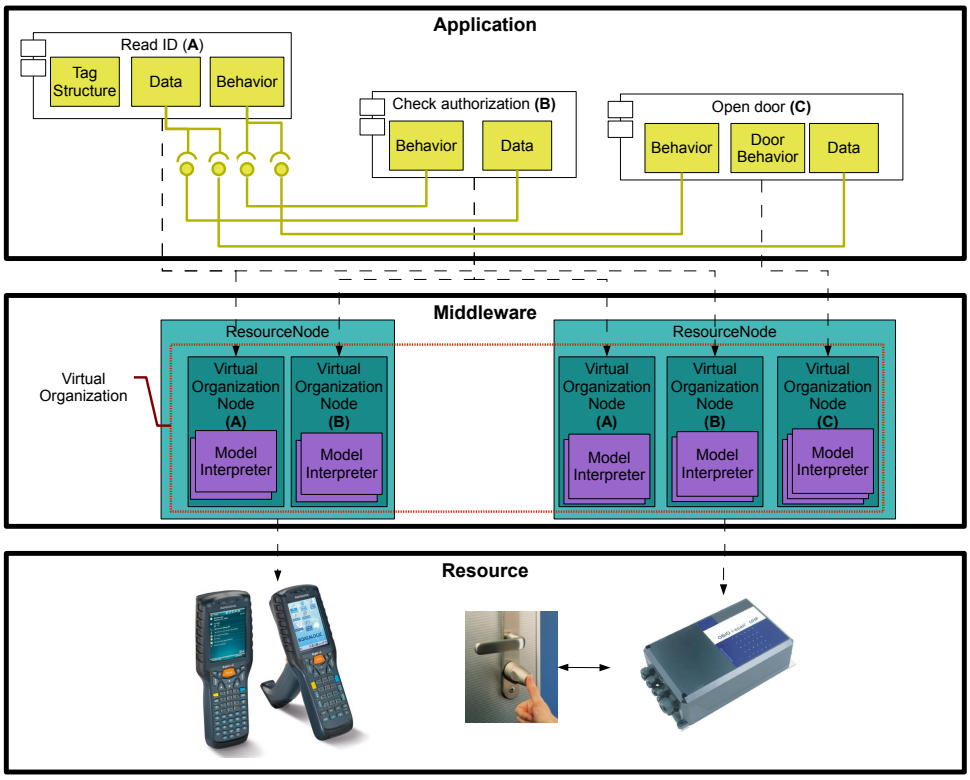


Figure 1.7: Example of layers

the assistance of mobile industrial handhelds processing this information via touchscreen or voice. The required applications also enable new business activities like on demand cooperation of partners in the supply chain by the connection of the executing devices with other (embedded) devices like RFID readers. This potential has been noted by software companies in the business domain of logistics, which provide solutions situated on different layers of Fig. 1.1. One company is Salomon Automation GmbH developing a warehouse management system (WMS), which is applied for controlling the business activities related to one or several warehouses storing goods. Another company is RF-IT Solutions GmbH promoting a middleware solution, which facilitates the usage of various RFID readers and tags in the business activities of companies. This emerging technology is used for identification of their products and for the exchange of data, which has been captured during the lifetime of these products.

In their currently provided WMS Salomon supports required mobile devices (e.g. industrial handhelds used by warehouse workers) as thin clients with respect to their software. The same observation holds for RFIT Solutions with respect to the targeted RFID readers by their middleware. As a consequence of this thin client approach, these devices are used for data capturing or data visualization purposes while the main application is targeted for desktop or server computing hardware. Beside the resulting requirements on a stable and performant network connection, their current solutions targeting mobile and embedded devices also provide very restrictive possibilities for reconfiguration by the user. Both

topics have been acknowledged by theses companies as important targets for their next generation of software products.

An exemplary application demonstrating the requirements for this next generation is depicted in Fig. 1.7. The main task of this application is the authentication of users based on ID tags and the control of an electronic door access system. If the presented ID tag is based on RFID technology, the authentication is done by a second component, belonging to another organization than the RFID reader. The organization owning the RFID reader is also in control of the door access system. If the presented tag is based on a barcode technology, the Read tag component is required to be executed on a mobile handheld equipped with a scanner. This component is also connected to the check authentication component, which can be executed on the mobile handheld or the RFID reader, while the door handling component requires the physical resource provided by the RFID reader, and is therefore not able to be executed on another device. To enable the communication between the components running on the mobile handheld and the components running on the RFID reader, their runtime nodes are required to form a virtual organization handling the access and execution rights of the components.

1.2 Outline of Thesis

A layered architecture has been introduced in the previous section and has been used for discussing current trends in application and middleware development in the form of Model-Based Development and Component Based Engineering. Furthermore the layered architecture has been utilized for demonstrating the heterogenic aspects in the software development by outlining the various organizational roles and different artifacts contained in each layer. Considering the fact, that mobile and embedded applications are designed increasingly upon distributed information, support for managing these systems is required. Such support is provided by the concept of Virtual Organizations, which have been introduced in the context of Grid systems. As emerging Grid types are also targeting the usage of mobile and embedded devices for managed personal access, the support of Virtual Organizations by the proposed architecture seems obvious.

In chapter 2 related work on application development with model based techniques as well as latest trends in component based software engineering are discussed. Objectives for this thesis are identified as a conclusion of the presented approaches. A runtime system targeting these objectives by enabling the interpretation of domain specific models on mobile and embedded devices is presented in chapter 3. Various parts of the runtime system are discussed and are associated with the publications presented in chapter 6. Case studies demonstrating the support of the proposed approach for mobile and embedded applications in the business domain of logistics are summarized in chapter 4. Finally chapter 5 provides a conclusion and gives an outlook on future work resulting from the insights obtained in this thesis.

Chapter 2

Related Work

Because software models are a central aspect in the proposed approach, their usage with respect to the different phases in the lifecycle of a software project is discussed in the beginning. In the second part existing component models and programming paradigms for mobile and embedded devices are presented.

2.1 Model-Based Software Development

Several definitions of models are compared by Muller et al. in their proposal of a graphical syntax to represent the relations between models in [MFB09]. According to Muller et al. the process of modeling aims to establish to represent something by something else. The usage of this representation as a a compromise of reality or as an caricature of reality is discussed by Jorgensen in [Jor09]. While compromising reality indicates the reduction of information from reality in a model for a specific underlying purpose, the caricaturing reality also emphasizes the overreaching exposition of specific information. Jorgensen finally noted that in both approaches models are build to improve the understanding of something complex (e.g. architects build a model of a building for demonstration purposes).

In software development projects the design pattern of layering [BMR+96] is used for dealing with complexity, by splitting the functionality of a program in several layers, with each underlying layer serving as the base for the next higher layer. A layered approach has been proposed by the Object Management Group (OMG) with the four-level meta-model hierarchy [Gro09c] for the definition of models in software development projects. In this standard each layer is defined by a distinct model, which can be used for the definition of the elements of the next layer. This model is called a meta-model with respect to the model of the next layer.

An example depicting elements and their relations in the various layers is depicted in Fig. 2.1 presenting the following layers:

M3: contains the base model for all modeling languages, the Meta Object Facility (MOF)

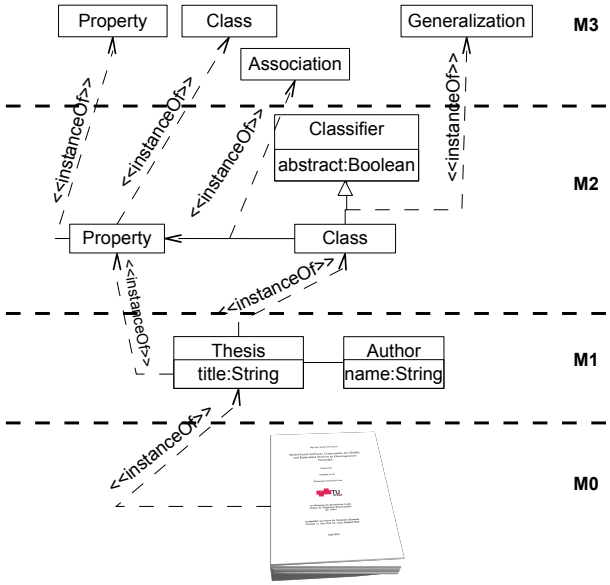


Figure 2.1: Elements in the layers of the four-level meta-model hierarchy

M2: is used for the definition of a specific modeling language like Unified Modeling Language (UML) [Gro09b] or the Eclipse Modeling Framework (EMF) [BSM+03]

M1: elements of this layer are defined by the software developer for modelling a specific software (like the definition of classes in Object Oriented Languages)

M0: holds the instances of elements defined in the software model in layer M1 (like object instances in OO languages)

Another definition of the meta-modeling approach is given by Kleppe in her discussion of software language engineering in [Kle08]. Her definition of a model is rested upon a combination of a type graph and a set of constraints at various types of this graph. A type graph is defined as a combination of

- a set of nodes which may include data types
- a set of edges
- a source function from edges to nodes, which gives the source node of an edge
- a target function from edges to nodes, which gives the target node of an edge
- An inheritance relationship between nodes (a reflexive partial ordering)

The concept of a labeled graph is also applied in the representation of a class diagram, which is usually made use of for the presentation of elements in the M3 and M2 layer of

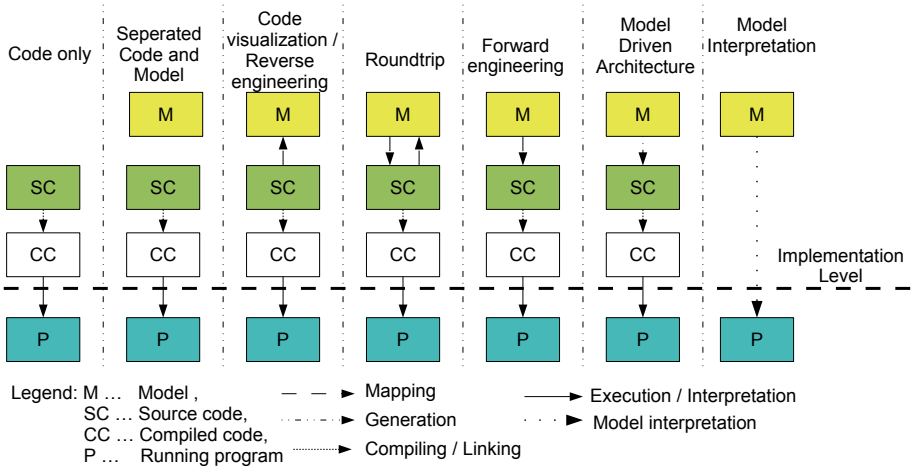


Figure 2.2: Different ways of applying models in the software engineering process (adapted from [VS06, p.74] and [KT08, p.5])

the OMG four-level meta-model hierarchy. Having motivated the usage of graphs for the definition of a model Kleppe also defines an instance of a model M as a labeled graph that can be typed over the type graph of M and satisfies all the constraints in M's constraint set. Having these definitions in mind, Kleppe introduces a meta-model as a model used to specify a language, while she also notes that there are various meta-models used in software language engineering (i.e. as an abstract syntax model of the specified language, a concrete syntax model and a semantic domain model).

While the usage of models for the specification of software languages is a newer aspect, their central role in a software engineering process is well established through Model Driven Software Development (MDSD)[VS06]. The idea behind MDSD is the separation of models containing domain specific knowledge from the information required for the technical realization platform. Transformations are applied on the model to retrieve a platform specific description in form of source code or binaries out of the modeled knowledge.

An overview about possible usages of a model in the software development process and applied transformations is given in Fig. 2.2. One extreme, as Kelly and Tolvanen noted in [KT08], is the specification of code without a model in mind or the missing automatic mapping between model and code. In the case of code visualization (a.k.a reverse engineering), round-trip engineering and forward engineering the used model and the code need to be at the same abstraction level as noted by Völter and Stahl in [VS06]. In contrast the generation step in Model Driven Architecture approaches is intended for providing an abstraction level in the model, i.e. the steps and knowledge required for creating a concrete representation out of the model elements is captured in the rules and templates of the generator. The other extreme is the direct interpretation of the model in an executing program, requiring the applied interpreter to contain the corresponding mappings between the abstract model and the concrete target platform. In all other approaches beside model interpretation the created target platform specific code is compiled and assembled with manually written program artifacts. The majority of MDSD projects is build

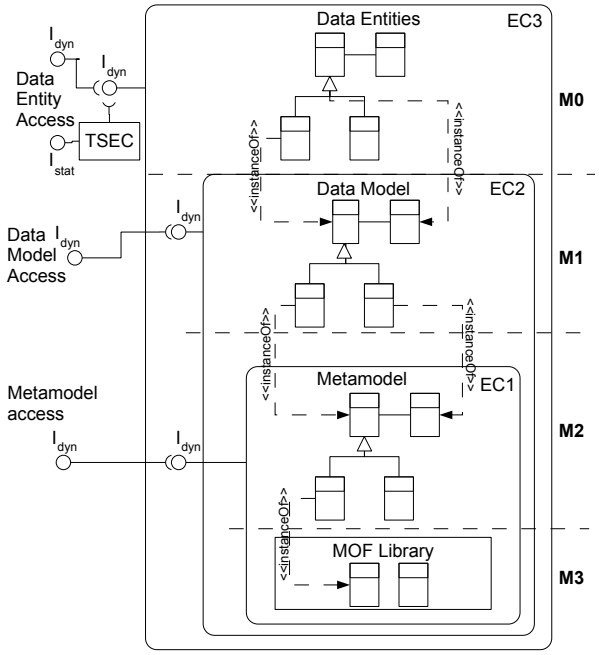


Figure 2.3: Four-level metamodel hierarchy implemented in the Entity Container (adapted from [Sch07, p.47])

upon a generative approach, with the transformations being specified in toolkit specific template languages. The former mentioned Model Driven Architecture (MDA) specification is thereby an MDSD approach based on the four level meta-model hierarchy and the Unified Modeling Language (UML) and has also been proposed by the OMG.

2.1.1 EntityContainer

Schmoelzer et al. have proposed the EntityContainer (EC) [SMK+05] as an object-oriented and model driven data cache to access data from a persistent data storage or to keep transient data during runtime. Communication with a data store is performed via the EC and its associated backingstores, which provide the mapping from the model to the corresponding data persistency technology (like a database or a file). As the EC is configured with the data model defined in a platform independent format and all the communication between datastorage and application is performed via the EC, switching the data persistency technology only requires the selection of the corresponding backingstore.

As depicted in Fig. 2.3 the data and corresponding models representing the different layers of the OMG meta-model hierarchy are stored in separate EC instances. Each EC instance of an upper layer (i.e. EC(x-1)) is used for validation purposes of data provided to the dynamic interface of EC(x). During initialization the meta-model and corresponding data model are loaded in EC(1) and EC(2) via their dynamic interfaces.

Structural diagrams	**Behavioral diagrams**	
		Interaction diagrams
Class diagram	Use Case diagram	Sequence diagram
Package diagram	Activity diagram	Communication diagram
Object diagram	Statemachine diagram	Timing diagram
Composition diagram		Interaction diagram
Component diagram		
Deployment diagram		

Table 2.1: Diagram categories in UML 2 [RQZ07]

In case of the data EC(2) also a static interface (a.k.a. Type Safe Entity Container [SKKT06]) is provided, which is generated out of the used data model and enables static type checking of the source code using the corresponding EC. The objects retrieved by the dynamic interfaces (Idyn) are entities with their own lifecycle and their own identity as defined Sect. 2.1.3.

Usage of the EC in the context of a Software Product Line (see Sect. 2.2.2) for implementing enterprise computing systems has been proposed by Schmölzer in [Sch07]. In the proposed method variability models as well as data models and user-interface models are applied in an EC based framework for dealing with different requirements of various customers of a warehouse management system.

2.1.2 Model views

While the EC applies all information contained in the data model for the creation and maintenance of the contained entities, typical modeling approaches make use of different views, with each view defining a specific subset of the model. An example of such an approach is the 4+1 view model of software architectures at system level developed by Kruchten [Kru95]. This model consists of different views on the described software architecture targeting different aspects and responsibilities:

Logical View: containing end user functionality

Development View: for programmers and software managers

Process View: for integrators describing performance and scalability

Physical View: for system engineers concentrating on the topology and used communication mechanisms

Scenarios: illustrating the architecture with selected use cases.

Each view can be specified using a specific modeling technique like textual or graphical modeling languages. One example of a collection of graphical modeling languages is the Unified Modeling Language (UML) specification by the OMG. UML is specified in the M2 layer of the four-level meta-model hierarchy and provides 13 different graphical modeling

languages (a.k.a. diagrams). These diagrams are distinguished in several categories which are presented in Tab. 2.1. According to Rupp et al. [RQZ07] they are split up in four compliance levels defined by the UML 2 specification. While diagrams for classes, types, packages and data types are contained in compliance level 0, the complete language definition of UML 2 is supported by compliance level 3. The introduction of these compliance levels is targeted for guiding UML tool developers and users, who should be able to select the right scope of the UML specification depending on the required diagrams. It can also be seen as an attempt to reduce the often criticised complexity of UML ([Gro09a, p.6]) in the same way as state-of-the art meta-modeling approaches rely on the definition of specialized models at the M2 layer.

2.1.3 Domain Specific Modeling

Domain Specific software development approaches with their focus on a distinct business domain have gained increased popularity in the last decade. One example relying on models has been proposed by Evans [Eva03] with Domain Driven Design (DDM). In DDM the usage of one model during the whole software development process is fostered. While in traditional software development processes different models are utilized for analysis, design and implementation purposes, one model is used during all phases of the DDM process. This fact facilitates the definition of an ubiquitous language for this project.

Various software design patterns are applied in the realisation of a DDM process. The model is accessed by repositories, while the realisation of modeled elements is encapsulated by factories. The expression of a model is realized with the following approaches:

Value objects: These model elements are defined by their attributes and do not need to provide an own identity, because they are not a central part of the modeled domain. Value objects are designed as immutable.

Entities: In contrast to value objects these elements represent important information in the modeled domain, thus requiring the definition of an own identity. This identity is defined by specific attributes and connections and is valid throughout the whole lifecycle of an entity.

Services: If an operation of a domain is not appropriate to be modeled as a value object or an entity (because it does not conceptually belong to this entity), this operation should be modeled as a standalone service interface defined in terms of other elements of the domain model.

In the discussion on Domain-Specific Multimodeling [Hes09] different realization possibilities for domain specific software ranging from single language programming through general purpose modeling to domain-specific multimodeling are distinguished by Hessellund. As the application of single language programming results in the masking of domain concepts in the syntax of a given implementation language, communication between software developers and domain experts becomes error prone and software evolution is aggravated. In contrast general purpose modeling as exemplified by the Model-Driven Architecture

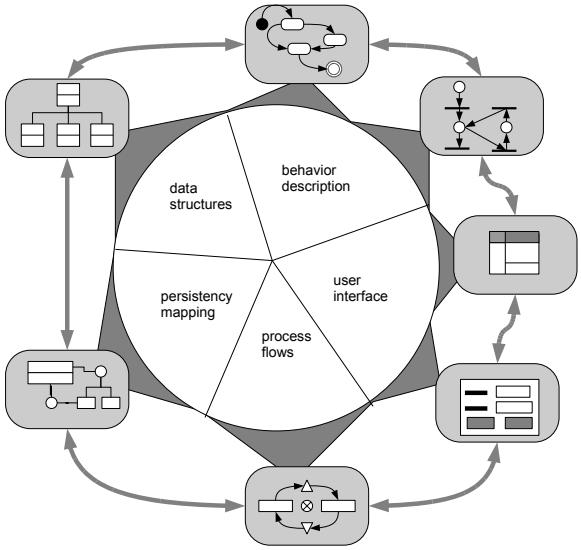


Figure 2.4: Multiple models for different domains (adapted from [Hes09, p.41])

and UML tackles the challenges of single language programming by providing the abstraction of domain specific concepts in a model, which is used as a first-class artifact in this approaches. A problem not solved by general purpose modeling is the clear separation of vertical and horizontal concerns of the modeled application, which is requiring the usage of different languages addressing the various domains as depicted in Fig. 2.4. The domains contained in the circle are tackling the horizontal concerns, which are used for the realization of applications of orthogonal business domains (e.g. warehouse management system, enterprise resource planning system or RFID aware applications).

2.1.4 Multimodeling within a domain

In contrast to the original figure presented by Hessellund, Fig. 2.4 has been adapted to visualize the possibility of more than one language for a given horizontal domain.

This observation is based on the discussion of different approaches for modeling the behaviour of software by Jorgensen in [Jor09]. Beside the depicted finite state machines (FSM) and petri nets, flowcharts and decision tables are also presented as possible behavioral modeling methods. FSMs consisting of states and transitions are used for structured analysis and realtime specifications. Events and actions specifying the behavior are associated with this states and transitions. Because modeling complex processes with FSMs leads to a larger number of states (a.k.a. state explosion problem) the FSM notation has been extended with additional elements in the proposal of (hierarchical) statecharts. One example of the application of statecharts is the Statemate approach [HP98], which relies on a combination of activity charts and state charts to model reactive systems. Another field of usage has been proposed by Horrocks to support the behavioral description of user interfaces [Hor99]. Petri Nets are consisting of nodes and transitions connected by

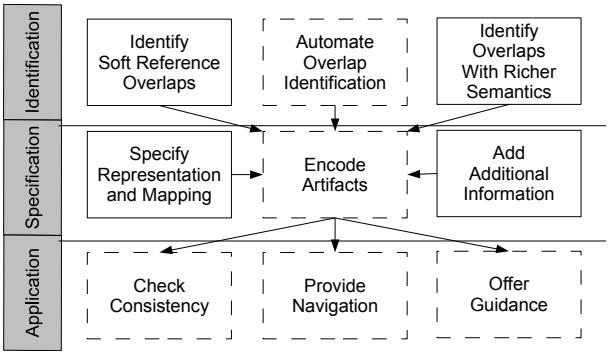


Figure 2.5: Coordination method for Domain specific multimodeling[Hes09, p.25])

edges, while Event Driven Petri Nets additionally make use of port input and output events. Flowcharts enable the graphical representation of a flow of events and decision tables are used for analyzing situations in which a combination of actions is taken under varying sets of conditions. Jorgensen has demonstrated with various examples, that none of the outlined approaches is applicable in all situations for the behavioral description of software.

Another example of different domain modeling methods is visualized in the user interface section of Fig. 2.4. This claim is based on observations from the development of the next version of the Eclipse integrated development environment (IDE). Initially started as a Java IDE, the OSGI based Eclipse framework has been extended with projects featuring the support of other programming languages or the development of models [Gro09a]. Additionally framework modules (like the user-interface module) are reused for application engineering in the Rich Client Platform (RCP) [ML05] approach. While the RCP approach is currently based on the usage of module specific programming APIs, specification of the application specific user-interface is going to be captured in a model in the next version of the Eclipse framework. While the application user-interface model is very coarse grained, the detailed layout of the contained dialogs and panels can be described with another modeling language as proposed by Traetteberg in [Træ08].

Based on the observed need for the use of multiple models in domain specific modeling Hessellund has proposed a coordination method depicted in Fig. 2.5, which is required for the successful development of multiple domain specific models in a single system. This method is made up of three steps. In the first identification step overlaps in the concern of two languages are identified. In the example of the two user interface modeling languages depicted in Fig. 2.4 this overlap is the integration of the specified dialog in the modelled application user-interface. The manually or automatically identified overlaps are used as an input in the second step aiming at the creation of a coordination model, which specifies how languages interact. The coordination model is an application specific artifact and is applied in the third process step for deriving the following development tool support:

- specification of consistency information to check the referential integrity of cross language references

- visualization and navigation in the different application specific models
- model based guidance of the developer (e.g. code completion based on the information from different models)

In their proposal of the experimental platform for multi-modeling NAOMI [DJS+09] Denton et al. have identified three key challenges for multi-modeling. For *capturing multi-model interdependencies*, which becomes difficult with the increasing number of applied models, these dependencies have to be made explicit. The second challenge is the *consistency of multi-models* as discussed by Hessellund, while the third challenge is the *semantic precision of inter-model data exchange*, which becomes difficult, if the different models are not specified in languages derived from the same meta-model. The NAOMI platform developed for tackling these challenges is made up of local components, providing a sandbox and execution engine for working with local copies of the used multi-models. These models are stored in a multi-model repository along with attributes defining the input and output of a model and constraints required to be satisfied by this model. Different models are linked by connectors, which are maintained by the local multi-model manager. According to the presented examples the different models are expected to belong to different meta-models and modeling approaches respectively.

2.1.5 Models at runtime

In conservative MDSD approaches software models are used at the development time of the system, featuring code generation based on the model specifying distinct parts of the system. In the context of UML several attempts are made to enable the usage of UML models at runtime. One technique is presented by Milisevic in the context of object oriented information systems (OOIS), by introducing a special UML profile and the corresponding runtime support. The OOIS UML approach as discussed in [Mil09] enables the specification of the data and behavior of an information system in a UML model, which may be transferred in a target programming language or may be interpreted. Both variants rely on the OOIS UML Model Library and runtime environment, which provides generic methods for creation and handling of the user interface. Milisevic also notes in his discussion of OOIS UML, that an Executable UML Foundation has been adopted by the OMG. A virtual machine is defined by a precise specification of a core subset of UML as well as a mathematical formalization of those semantics.

Beside these UML based approaches Bencomo [Ben08] has motivated the usage of models@run.time to check the correctness of the currently executed system by supporting the monitoring and adaptivity of applications. The following definition of a model@run.time is given by Blair et al. : " A model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective."[BBF09, p.23] According to this definition a runtime model is a "live" development-model, which enables dynamic evolution and the realization of software designs.

Cetina et al. [CGFP09] are discussing the usage of variability models at runtime to support a model-based reconfiguration engine in the context of a smart home system. A runtime

architecture is proposed by Morin et al. based on the usage of models at runtime for supporting a dynamic software product line in [MBJ+09]. Their approach relies on four meta-models containing

- the systems variability,
- the context variables of the system environment relevant for adaptation,
- the rules for selecting the features according to the context and
- the description of the component-based architectures.

While the latter approaches are focused on the reconfiguration of component based architectures, taking the specification and implementation of the components as granted, the OOIS UML approach presented by Milicev is not emphasizing the split up of the used models in different components as well as a modular design of the model runtime environment.

2.2 Component Based Software Engineering

Crnkovic et al. note in their discussion of basic concepts of software components [IC02], that there are several definitions of a software component available. The following definition given by Szyperski in [Szy02] is also used as the basis for their discussion of the different definitions:

Definition: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party".[Szy02, p.41]

Crnkovic et al. also refer to the varying definitions of a software component in industry and academia. While industrial software components are seen as a well-defined entity with often small and easily understood functional and non functional features, the academic definition of a component is more coarse grained. This difference has been targeted by the introduction of hierarchical software components in academic approaches, which are made up of a composition of already existing software components.

Handling of software components is specified in a software component model. A taxonomy of various software component models is given by Lau and Zheng in [LW07]. This taxonomy is based on the syntax, semantics and composition of the components based on the given component model.

According to the syntax the following categories are defined:

OO: object-oriented programming language

IDL: programming-languages with IDL mappings

ADL: Architecture Description Languages (such as UML components)

While component models in the semantic category distinguishes are differentiated by their handling of the defined components, four categories of component composition are distinguished by Lau and Zheng.

Category 1: New components can be deposited in a repository but cannot be retrieved from it by the developer. Composition of components is not possible in the design phase, but by an assembler in the deployment phase.

Category 2: Components are stored in a repository and can be composed in the design phase, but the composition rules are stored in the repository instead of one composite component. As a consequence the composition can not be changed during the deployment phase, i.e. the single components are transferred and instantiated based on the specified composition rules.

Category 3: Components can be retrieved from the repository as well as composite components can be stored in the repository. No changing of this composition is allowed in the deployment phase.

Category 4: Components are not stored in a repository, i.e. they are all constructed and composed from scratch. In the deployment phase no new composition is possible.

The results of the taxonomy according to the criteria defined above are presented in Tab. 2.2. The compared component models are partially discussed in [LW06] and the following sections. Note that in Tab. 2.2 different component models are also categorized depending on their application area as proposed by Teiniker in [Tei05]. While enterprise computing components have higher requirements on storage and performance of the target platforms than desktop computing approaches, some approaches can be used in several areas depending on the mapping used in the realization. Some approaches are used for specification of software components in embedded system (e.g. consumer electronics), while other component models are targeted for pervasive systems targeting mobile and embedded devices.

The implementation and assembling of components is often supported by an execution environment specific for the chosen component model. Depending on the applied implementation approach for this execution environment, Szyperski is differentiating a component framework and a component platform.[Szy02, p.548f] A component framework is defined as a collection of rules and interfaces (contracts) that govern the interaction of components. Component frameworks can be organized hierarchically being components themselves. This fact is the key difference to a component platform , which is the foundation for components to be installed and executed on. A component platform typically contains an execution environment and provides additional services.

2.2.1 Related software component models

This section gives an overview of selected component models from Tab 2.2, which are relevant for the approach in this thesis.

Name	Target environment				Syntax	Composition
	Server	Desktop	Mobile	Embedded		
EJB	+	~	-	-	OO	2
CCM	+	~	~	~	IDL	2
Java Beans	~	+	-	-	OO	1
COM	~	+	-	-	IDL	2
.NET	~	+	+	+	IDL	2
Fractal	~	+	+	+	IDL	4
GCM	+	+	+	+	ADL	3
SOFA-2	+	+	~	~	ADL	3
OSGI	+	+	+	+	ADL	4
Kobra	~	+	-	-	ADL	3
MARMOT	-	-		+	ADL	3
BCF	+	+	-	-	ADL	3
Koala	-	-	-	+	ADL	3
Pin	-	-	-	+	ADL	4
PECOS	-	-	+	+	ADL	4
Jadabs	-	+	+	~	ADL	4
OpenCom	~	+	+	+	IDL	4

Table 2.2: Taxonomy of software component models (based on [LW06, Tei05])

.NET

The .NET framework is proposed by Microsoft implementing a simplified component model for deployment purposes in the form of assembly files bundling several corresponding classes. A component runtime is specified with the .NET Common Language Runtime(CLR), which is implemented for the Windows operating system for different hardware architectures. Another implementation is provided by the Mono project, enabling the execution of .NET applications on other operating systems. Depending on the target architecture different specifications of the .NET framework are available. While the full *.NET framework* is targeted for desktop computing purposes, it can also be applied for server computing applications. On the other side the *.NET Compact framework* is specified for handheld devices running a Windows CE based operating system. The *.NET Microframework* has been introduced in the last years for embedded devices with low memory and energy constraints like sensor nodes or small displays. Applications following this specification are intended to be executed directly on the hardware or with a basic support of an operating system through a hardware abstraction layer.

Another similar approach based on virtual machine technology, which is also very popular for industrial applications is the Java platform initiated by Sun Microsystems. The initial specification does not provide a component model, but is also leveraging a simple modularization technique for deployment purposes by distributing a collection of executable files bundled in compressed archives. For enterprise and server computing purposes the specifications of Java Enterprise Edition (J2EE) and Java Standard Edition (J2SE) are defined, while the development of applications with reduced functionality for mobile and

embedded devices is enabled by two configurations available for the Java 2 Micro Edition (J2ME). The Connected Device Configuration (CDC) is targeted for handhelds, whereas the Connected Limited Device Configuration (CLDC) is specified for very resource limited devices (e.g. mobile phones).

OSGI

The OSGI component model [All09] has been specified to support the component based development of Java applications. It is realized by a runtime environment, which is executed on top of a Java virtual machine. OSGI components are developed as bundles providing services and extension points. These bundles can be used by other bundles being plugged into the OSGI runtime environment.As the OSGI reference implementation is used as the basic framework for the Eclipse IDE and because of the popularity of the RCP concept (discussed in Sect.2.1.4), this component model has become quite popular in industry in the last years.

Business Component Factory

While the Business Component Factory [HS00] (BCF) approach of the OMG is not primarily targeted for mobile or embedded devices its separation of the functional aspect from the technical aspect of a software component is an important feature related to the approach presented in this thesis. Based on this separation different components are defined in this approach:

- Distributed Component: is the most fine-grained form of a component, which is implemented following any industrial component model.
- Business Component: This component implements a single autonomous business concept and usually consists of one or more distributed component instances. A business component can be deployed across several machines.
- System-level component: The largest grained component contains various business components with clear defined interfaces to realize a specific business need.

Grid Component Model

The Grid Component Model (GCM) [BCD+09] has been proposed by Baude et al. in the area of Grid Computing. GCM is extending the Fractal component model while treading components in a more coarse grained way. This attempt is required for facilitating the construction of adaptable and autonomous components, which is also supported by the abstraction of runtime resources through the concept of virtual nodes and deployment descriptors. These aspects are used in an iterative automatic mapping algorithm trying to find a feasible deployment of the required components on the given runtime environment. Additionally specific interface types meeting the requirements for multiway communication in grid computing are supported by this approach. Finally the handling

of non-functional(NF) properties is refined in contrast to Fractal. In both component models controllers are responsible for handling the various NF properties by forming a membrane around the managed component. To support the evolution of the execution environment this controllers are additionally designed as pluggable components, enabling a better adaptability in GCM compared to Fractal.

SOFA

The SOFA approach is specified by a hierarchical component model [BHP06], with two types of components. Composition is performed by the first type, which consists of a black box view of the component and a gray-box view defining the composition at the first connection level. Composition is done by connecting the provided and required interfaces of other composite components or primitive components. This primitive components represent the second component type in SOFA and are implemented by a component controller and the platform specific implementation of the component.

SOFA components are specified in an ADL file, which serves for generation purposes of the component implementation. After the implementation they are stored in a repository, where they are loaded from by the SOFA runtime environment. Addressing several challenges in component systems, such as dynamic reconfiguration support, explicit access of the controller functions in the component implementation and support for multiple communication styles, the succeeding SOFA-2 component model has been developed. A model based approach is used in this component model for generation of the component repository and the corresponding tool suite. The models are also used for the generation of code skeletons for the implementation of the primitive components, for deploying the components and for setting up the application.

The runtime environment is called a SOFAnode, which consists of several deployment docks (a SOFA container and the corresponding virtual machine) and a repository for retrieving the required components. While the component model and the used communication approach is platform independent, the platform specific implementation of the domain specific functionality of the primitive components makes the reconfiguration of the component application still difficult for not supported target platforms.

Jadabs

The Jadabs component model has been proposed by Frei in his PHD thesis [Fre05]. Fig. 2.6 depicts the relationship between Jadabs and other component models. Like the layered architecture of distributed systems proposed by Tanenbaum (presented in Sect. 1.1), low level services are realized by a platform while high level services are provided in a component model. Like in the work presented by Teiniker several component models for infrastructure environments have been considered, but Jadabs has been also compared to other component models for mobile environments. Jadabs empowered the possibilities of a service-oriented architecture and defined two types of runtime containers, namely a segmented-adaptive container and a monolithic-dynamic container. The first one allowed a transparent adaptation of components with dynamic loading and unloading, while the

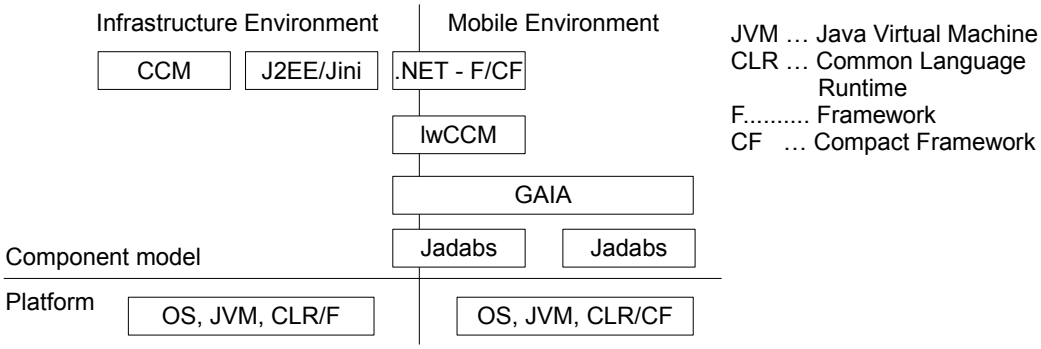


Figure 2.6: Classification of mobile component models [Fre05]

second one required the components to explicitly expose the interface for the adaptation. The implementation of Jadabs leverages a service oriented architecture defined by the OSGI specification (introduced in Sect. 2.2.1) and applies an XML mapping for the platform independent communication mechanisms. Adaptation of components is done with the application of aspect-oriented programming techniques. As a consequence the modification of components is realized at source-level. This level has been used for the implementation of the services, which contain the domain specific knowledge.

OpenCom

Coulson et al. proposed the OpenCom component model depicted in Fig. 2.7 in [CBG+08]. The proposed framework addresses the requirements of target domain independence, deployment environment independence and is designed for a negligible overhead. These issues are resolved by the component runtime kernel, which is responsible for loading and binding components. This kernel is enhanced by extensions, which are themselves implemented as components and are used for the realization of target domain specific requirements. The extensions are applied in the realization of component frameworks, which are defined in a coarser granularity as components. Component composition is used to address some focused area of concern or to implement a well-defined extension protocol accepting additional plug-in components. OpenCom components are instantiated inside a capsule providing access to the kernel and supporting composition of other component instances by connection of the corresponding interaction points (namely interfaces and receptacles).

PECOS

The PECOS component model is introduced by Nierstrasz et al. in [NAD+02] and is applied for the specification, composition, configuration checking and deployment of soft-

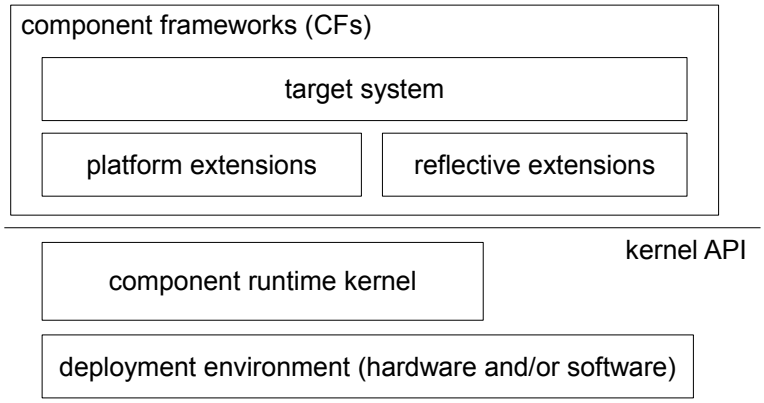


Figure 2.7: Layers in OpenCom

ware components on field devices. Field devices are characterized as reactive, embedded systems, which make use of continuously data gathering sensors to control connected actuators, valves or motors. Hierarchical composition of PECOS components is supported, where a PECOS component can be (i) a passive (i.e. has no own thread of control), (ii) an active (used for modeling longer-lived activities) or (iii) an event component (used for hardware pieces frequently emitting events). Components are connected by ports and have a number of property bundles. Additionally a behavior is specified for each component, consisting of a procedure handling the data available on the ports and producing effects in the physical world.

MARMOT

The MARMOT methodology is an adaptation of the KobrA [ABB+01] approach, which is based on the usage of multiple UML diagrams for the specification and realization of components. In the KobrA method (depicted in Fig. 2.8 these implementation models are used for the transformation in a tool specific format (i.e. source code), which can be automatically compiled into binary executable files; as a consequence the whole model information is only available at system development time. While the KobrA methodology is targeted for desktop and server computer systems, the MARMOT approach is used for the specification of components running on embedded devices. In the discussion of interaction consistency checks in MARMOT component refinements a framework is introduced by Choi in [Cho07]. This framework is based on the Kobra methodology and enables a transformation of the UML specific implementation models (e.g. class diagrams and statecharts) into consistency models (e.g. PROMELA of the SPIN model checker). Sabil and Jawawi discuss the integration of the PECOS component approach into MARMOT in [SJ09], presenting a process for analyzing and designing components of embedded real time (ERT) applications with MARMOT. These components are then mapped for implementation to the PECOS component model by generating corresponding code skeletons, which are extended to produce the target system.

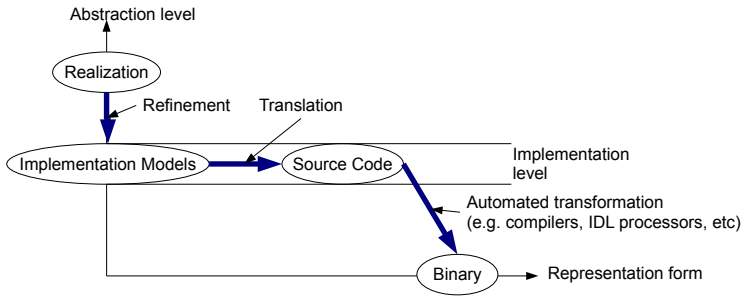


Figure 2.8: Steps and involved artifacts in the KobrA method[ABB+01]

2.2.2 Variability in component frameworks

Models in the context of software components can be used for the specification of the component and for the definition of the component framework. The second aspect of model usage targets the domain specific definition of a component framework to cope with the heterogeneous constraints of the developed software components. The proposed approaches make use of variability models as defined in the context of Software Product Line Engineering (SPL) . A definition of SPL is given by Pohl et al.:

Definition: "Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisations."[PBvdL05, p.14]

Mass customisation is typically provided by the selection of relevant features to tailor a large-scale product to individual customer needs. The available features for selection are specified as part of traditional software models or in a distinct model. Such a model is defined by the orthogonal variability model, "that defines the variability of a software product line. It relates the variability defined to other software development models such as feature models, use case models, design models, component models and test models."[PBvdL05, p.75]

While the first aspect is increasingly used in modern component frameworks (e.g. SOFA2 discussed in Sect. 2.2.1), it is also supported by a specific component diagram in the UML. Cheesman and Daniels [CD00] have proposed a process for the specification of component-based software in the context of UML, which is based upon the usage of the various UML diagrams for the definition of the contracts regarding the usage and the realization of the corresponding components.

Targeted at the OpenCom component model discussed in Sect.2.2.1, the Genie approach for generating middleware families has been introduced by Bencomo in [Ben08]. Several levels of abstraction are provided to handle two dimensions of dynamic variability (i.e. structural variability, environment and context variability). These variabilities are specified using domain specific modeling languages (DSMLs) serving as input to generators for components, component configurations and reconfiguration policies. Such a generated domain specific middleware family makes up a middleware framework, which is represented by different configurations at runtime depending on the selected variabilities.

Another approach implementing a meta-component system for the SOFA component model (see Sect.2.2.1), has been presented by Bures et al. in [BHM09]. This meta-component system is configured using SPL techniques to create a component system out of core assets defined in the solution space. The component system is used for the creation and management of corresponding software components by providing the specific development, design and deployment tools as well as an execution environment.

2.2.3 Component support in Event Based Systems

Another approach fostering the decoupling of components in distributed systems has been proposed in the context of Distributed Event-Based Systems. Mühl et al. give the following definition of an event-based system:

"In an event-based mode of interaction *components* communicate by generating and receiving *event notifications*, where an *event* is any occurrence of happening of interest, i.e. a state change in some component. The affected component issues a *notification* describing the observed event. An *event notification service* or *publish/subscribe middleware* mediates between the components of an *event-based system* (EBS) and conveys notifications from *producers* (or *publishers*) to *consumers* (or *subscribers*) that have registered their interest with a previously issued *subscription*." [MFP06, p.3]

To limit the visibility of events between different components forming an EBS, the notion of scopes is proposed by Mühl et al. by the definition of a meta-model enabling the bundling of a set of components in a scope. Note that a component can also be member of multiple scopes, which leads to the usage of an acyclic directed graph containing a set of components and the relationship between these components. Furthermore different types of interfaces are defined for scopes, based on the filtering mechanism of ingoing and outgoing interfaces defined by simple components of an EBS. While these interfaces are contained in a base Interface I_C, additional scope specific filters can be applied to filter messages exchanged between two components in a scope. Depending on the placement of the filters on the beginning or end of an edge, which connects these two components in a scope graph, the interface is named a selective interface or an imposed interface. The combination of both interface types together with the base interface constitutes the effective interface, defining which notifications are exchanged between two components.

2.3 Summary

In conclusion of the related work presented in the last sections Tab. 2.3 provides a taxonomy of the different approaches regarding the objectives of this thesis. According to this taxonomy current component models are supporting mobile and embedded devices and are usable for distributed organizations. While models are used for the specification of component interfaces, the implementation of a component is principally provided by a code artifact capturing the domain specific knowledge. These artifacts can be created using model-based development techniques, typically applying code generation to produce skeletons, which are extended by the developer. On the other side the usage of several

Approach	Multiple domain models	Transient Model Extension	Components	Mobile and Embedded support	Supports application distribution	Evolution Compatibility
Entity Container	partially (data)	no	partially [STK08]	no	partially (Backingstore)	yes [STK08]
Domain-Specific Multimodelling	yes	no	no	no	no	no
Business Component Factory	no	no	yes	no	yes	partially
.NET	no	no	yes	yes	no	no
OSGI	no	no	yes	yes	yes	no
Jadabs	partially	no	yes	yes	partially	no
SOFA-2	partially [BHM09]	no	yes	yes	partially [BHP+07]	no
OpenCom	partially [Ben08]	no	yes	yes	partially [CBG+08]	no
MARMOT	yes	no	yes	yes	no	no
PECOS	no	no	yes	yes	no	no
Event Based Systems	partially [MFP06]	no	yes	yes	partially [MFP06]	no

Table 2.3: Taxonomy of related work based on objectives of thesis

models covering various application domains has been proposed in literature, to tackle the problem of incomplete specification of the model. As a consequence the model is able to be interpreted at system runtime, enabling an adaptation of the system structure and system behavior based on the problem space. As these approaches are typically running on devices providing enough resources, separation and bundling of different models is not a key aspect of these approaches instead the models are treated as a whole entity. On the other side the applied metamodels are treated as separate entities, thus requiring a management infrastructure for each of these metamodels.

The EC approach has proved useful for managing models based on a data metamodel, and has also been applied for the definition of model typed component interfaces. As the EC reflects a layered metamodel hierarchy the usage of other metamodels is feasible.

2.3.1 Objectives of this thesis

Based on the challenges discussed in Sect.1.1.4 and the concluded related work presented in the last section, the following objectives for this thesis can be derived.

Objective 1: Support bundling of multiple models containing the complete specification of a problem specific functionality and foster reuse of these bundles across organizational boundaries

MultiModeling aims at the application of different models containing various views of the modelled functionality. Anyway separating these models in several parts and bundling of various model parts according to the logical organization of the modelled software is not targeted by such approaches. In contrast the proposed methodology enables the specification of components at the model level. For deployment and reuse purposes the bundling of several partial models is treated as one artifact. An architecture of a system for the specification and execution of such multimodel-based artifacts is required. Furthermore the reuse of these artifacts for the development of new functionalities should be supported by this architecture.

The challenge of artifact heterogeneity in each layer as motivated in Sect. 1.1.4 is tackled by this objective. The realized system should be based on existing component technologies, which provide mature techniques for the specification of static interfaces. Such static interfaces are a consequence of the application independent and generic definition of the system specific components. The portability of the proposed system on other platforms is eased by this fact.

Objective 2: Enable dynamic configuration of the runtime system respecting resource constraints

The combination of several models (based on distinct meta-models) used for the implementation of a specific component determines the required plug-ins of the model interpretation architecture, which are loaded at execution time of the component. Furthermore the meta-model of the component can be shared by the model specific plug-ins. Both aspects are supporting resource constraints typically to mobile and embedded devices. This approach provides a bridge between the static meta-model used in current Model Driven Software Development approaches and the separated meta-models of current MultiModelling methods.

Objective 3: Enable shared usage of resources

A clear separation of concerns between resource owner and resource user should be supported by the proposed system. This requirement is tackled by the concept of Virtual Organizations, which are made up by several members of the runtime environment. The resource access methods are contained in the action statements of the used models. This enables the validation of granted resource rights to the component by verification of the specific models at runtime.

2.3.2 Contributions of this thesis

The following contributions are claimed as outcomes of the research in this thesis regarding the objectives presented in the previous section .All contributions have been evaluated by

prototypes during the case studies summarized in Chapter 4 as well as in the specific papers presented in Chapter 6.

Contribution 1: Model-Based Software Components supporting the bundling and distribution of multiple models

The concept of Model-Based Software Components (MBSC) is proposed in the publication available in Sect. 6.1, enabling the definition of a problem specific functionality with multiple models realizing different views on the specified component. Furthermore the concept of MBSC connectors specified in the various models is used for combining different MBSCs for the realization of new functionality. For specification of actions and queries in the different models of an MBSC, a domain specific language named ECQL is proposed the publication available in Sect. 6.2. The distribution and management of models in a Mobile Grid environment has been evaluated for data models in the publication in Sect.6.3.

Contribution 2: Transient Model Extension enabling a plug-in based runtime node

The concept of MBSCs has been enabled by the theory of Transient Model Extensions (TME) presented in Sect. 6.4 and Sect. 6.5, allowing the dynamic extension of a meta-model to enable the specification and validation of corresponding model elements at runtime. This aspect has been considered in the construction of a plug-in based runtime node as discussed by the publication in Sect. 6.6, where all plug-ins share a dynamically assembled meta-model defined by the executed MBSC. The primary mechanism of interface compatibility for connecting several of these runtime nodes is presented in the publication available in Sect. 6.7.

Contribution 3: Additional nodes enabling the shared usage of resources

To provide a separation of concerns between the executed MBSC and the used resources, the nodes discussed previously have been embedded in a distributed runtime system. They have also been extended with management support for virtual organizations realized by this runtime system as discussed in Sect. 6.8. Furthermore the TME approach has been applied to extend the ECQL language with resource specific methods. Each runtime node targeted for executing an MBSC is checked for the existence of these required methods during the MBSC instantiation. In case of a missing method the execution of this MBSC is circumvented by this runtime node.

Chapter 3

Design of a runtime system for MBSCs

As a tribute to the increased dissemination of mobile and embedded devices, their role in the realisation of business processes gets more important. A mobile environment is formed by all devices contributing to this business process, with each device defining a target platform at several layers as noted by Atkinson and Kühne [AK05]. Model Driven Development allows to deal with these heterogeneity in the target platform at system development time, with the state-of-the-art approach of code generation for the executable artifacts. This approach hampers the reconfiguration support of these artifacts, because the implementation is bound to a target platform at the development time.

The approach presented in this thesis makes use of the models at runtime of an application by implementing a model aware middleware, which enables the specification of components and their assemblies by models. These models are interpreted by a distributed plug-in configurable runtime environment.

Fig. 3.1 gives an overview of the layers involved in this approach based on the layered model introduced in Sect. 1.1. Like in traditional MDSD concepts a platform specific and a platform independent layer (i.e. application specific layer) is defined. On the left side the publications written during the creation of this thesis are presented, which have highlighted the following topics.

- *Model Based Software Components*
  For developing applications in a platform independent way a model-based development approach is presented in Sect. 6.1, which is based on the interpretation of a set of models, providing different views on one software component – like state machine and class diagrams. Additionally the integration of components build with this approach in the design of a mobile information system is presented.

- *Usage of data models at runtime of mobile grid environments*
  The need for managing data access in mobile applications, regarding concerns of privacy and data integrity is discussed in Sect. 6.3. By relying on a mobile grid

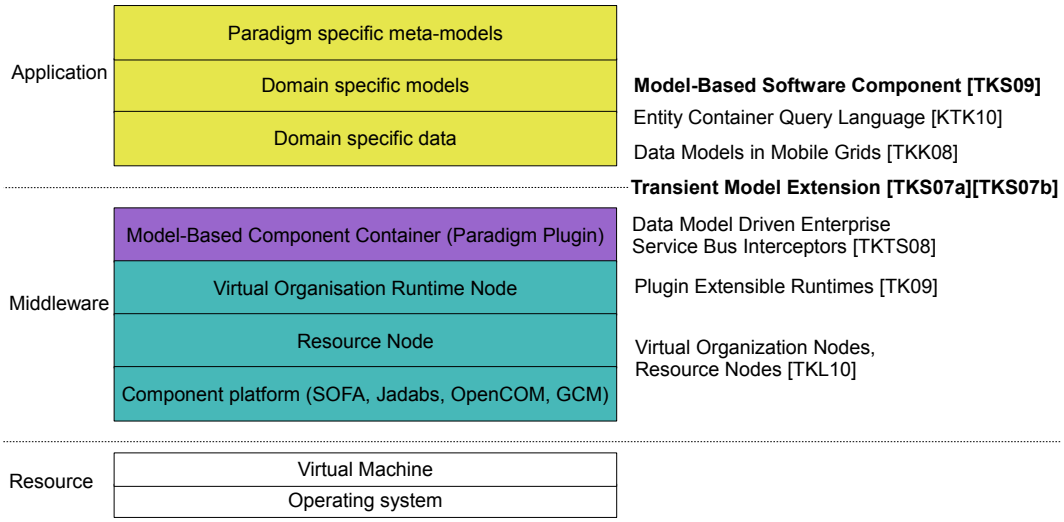


Figure 3.1: Layered architecture of proposed approach and publications corresponding to the different layers

infrastructure, the central concept of a virtual organisation can be extended to enable the exchange of data models between the members of this VO, containing the externally accessible data structures of the publishing member. By extending the model-based approach for data access in a data intensive system, mobile devices are enabled to dynamically use data of a mobile grid system,

- *Transient Model Extension*
  The mechanism of Transient Model Extension (TME) is discussed in Sect. 6.5 and Sect. 6.4 recognizing the fact that software is often constructed using a layered approach to encapsulate the functionality in different layers. Individual requirements of each layer demand layer specific data structures, which typically provide redundant information with respect to the data source. A mechanism for transient extension of a data model is presented, allowing a basic data model to be used by every layer, being extended by additional attributes and classes for satisfying layer specific requirements.

- *Resource node and runtime node architecture*
  In Sect. 6.6 and Sect. 6.8 a pervasive runtime architecture is presented that explicitly honors the ownership and realm of control of hardware devices, computing resources, device-connected I/O resources, and application components. Based on such an architecture, their owners, while pursuing their very own business models, can cooperatively form and take down Virtual Organizations (VO) to run applications in the sense of Grid Computing and Utility Computing. Heterogeneous platforms supporting runtime reconfiguration are connected with the help of a portable, plugin-extensible runtime environment, which is able to run model-defined software

components (MBSC). MBSCs consist of a set of high-level models for classes, state machines or user interface descriptions that are directly interpreted by these architecture's runtime nodes. To enable also the model-based specification of applications a domain specific language has been proposed, which enables the specification of the access and manipulation statements for data entities provided by the various MCCs in a model.

- *Enterprise Service Bus Connectors based on model compatibility*
  Integration of various software systems is an important issue in the execution of distributed business processes. Assembling of loosely coupled services via XML based protocols is a frequently used technique today. To overcome the struggle between safety of a strong typed interface and flexibility of generic parameters, an approach is presented in Sect. 6.7. This approach used model-typed interface parameters together with the idea of model compatibility verification. Separated ownerships of service provider and consumer interfaces has been respected and a mediating connector based on platform-independent, model-based functional interface reconciliation has been introduced. This approach could be used in the proposed runtime system for coping with differences between connected VO Nodes.

3.1 Model-Based Component Container

The concept of the Model-Based Component Container is based on the four-level meta-model hierarchy proposed by the OMG (see Sect. 2.1) and the lessons learned from the application of this hierarchy for data models in the EntityContainer (EC) approach (discussed in Sect. 2.1.1).

As proposed in the EC approach (presented in Fig. 2.3) the elements of the model layer M0 (containing the application data) as well as of the model layer M1 (containing the application model) are each managed by a separate EC instance. The static layer M3 specifies the basic meta-model and is shared by all MCC instances. The layer M2 provides an MCC specific meta-model, which defines the model paradigm represented by this MCC instance. The information provided by this layer is also used by a controller interpreting the application model M1 for manipulation of the application data M0. The introduction of this controller is the major extension of the original EC concept, enabling the application of the MCC on models representing other information than data (e.g. user interface or behaviour).

3.1.1 Transient Model Extension

Another addition to the original EC concept as proposed by Schmoelzer et al. [SMK+05] is the technique of Transient Model Extension (TME) discussed in Sect. 6.4 and Sect. 6.5. The key idea of the TME proposal is the dynamic extension of a model managed by an EC, to enable the creation of new data elements in the next higher model layer. Initially this concept has been proposed for the extension of the data model, which also specified the structure of the persistent storage. This technique enables dynamic attributes, which

are not managed by the persistent storage. An exemplary application of this approach is the addition of an additional attribute containing the age of a person, which is based on the difference between the birthday provided by the persistent datastore and the current date provided by the framework of the EC.

An additional benefit of the TME approach is the possibility to register callbacks, which are executed by the EC infrastructure upon the creation and update of an TME defined element. This mechanism can be applied to augment the model with platform specific code.

As the TME can be applied for the creation of any model element defined by the corresponding metamodel and its application does not effect the original model used by the EC, it can be also applied to merge two different models (which is similar to the approach of package merge as specified by the UML [Gro09c]). While the UML package feature is mainly used at the runtime of the corresponding development tools for the software model, the TME approach can also be used to support the execution of models at the runtime of the modelled application.

3.1.2 Model View support

The usage of different model views to light specific parts of an application has been discussed in Sect. 2.1.2. According to the previous section, the EC approach was initially proposed to handle data entities, which are retrieved and created for a persistence storage. A specific data model is used by the EC and for the creation of this persistence storage. Aiming at the model-based specification of a whole application requires the support of various model views by the realizing framework, which can be sometimes applied for the same domain. Exemplary model views for the specification of application behavior are state machines or petri nets (as suggested by Jorgensen [Jor09]). For the definition of a specific user interface different model views can target an interaction via a graphical or textual interface. Following the OMG meta-model hierarchy approach each modeling view is specified in a specific meta-model managed by the M2 layer of an MCC.

Additionally a view specific controller can be defined mapping the given information of the model at layer M1 to data at layer M0 and to corresponding platform specific instances. A corresponding meta-model targeting the definition of data, user interface and behavior (via a statemachine) of an application running on mobile handhelds is proposed in Sect. 6.1. The model layer organization of different MCCs is discussed in Sect. 6.6. It could be demonstrated that the data and model layer of each MCC are managed by distinct EC instances while the M2 model can be constructed by merging all specific M2 models of the different MCC types. As a consequence this M2 model can be provided by one EC instance, which can be shared among the different MCCs. This technique reduces the memory consumption of the overall approach, which addresses the memory constraints of mobile and embedded devices.

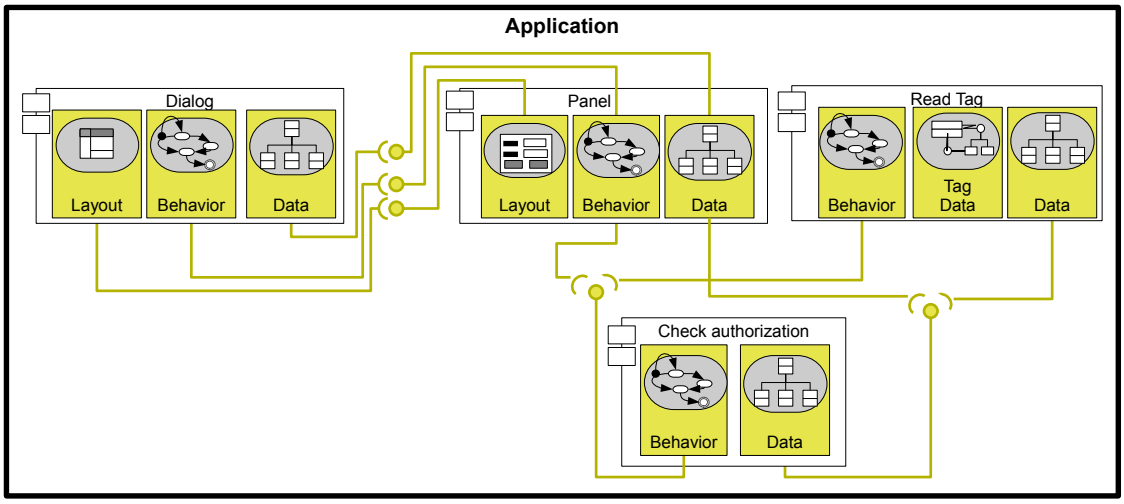


Figure 3.2: MBSC models and connectors

3.1.3 Entity Container Query Language

A runtime environment consisting of several MCCs has been introduced in the last section to cope with various models representing different views on the modelled application. To enable a full definition of the application at the model level, a mechanism is required for specification of interaction between these different views. One example of such an interaction is the binding of data specified in a data model to a user interface control specified in the user interface model. Another example is the manipulation of data or the user interface as a consequence of an event or a state change in a statemachine. Also the retrieval of data values for checking certain conditions in a state machine is an example of such an interaction.

These requirements are targeted by the Entity Container Query Language (ECQL), which is discussed in Sect. 6.2. ECQL is defined as a Domain Specific Language for formulating action and queries to be executed on an MCC and is based on a specific metamodel as suggested by Kleppe [Kle08]. In contrast to existing proposals for action and query languages (like OCL used for specification of constraints in models or OQL which enables the definition of queries based on a data model) the language elements of ECQL can be extended at runtime via the TME mechanism, because ECQL statements are also managed by the EC infrastructure. This aspect is used in the runtime platform to specify the usage of shared resources in a model as discussed in Sect. 3.3.1.

3.2 Model-Based Software Component

Based on the presented MCC concept and the insights from the research on Model Typed Component Interfaces [STK08] the concept of Model Based Software Component (MBSC) has been proposed. This concept is demonstrated in Fig. 3.2, which depicts different

MBSCs. These MBSCs are defined by multiple models. As visualized by the icons of the different (meta)models (presented in the discussion of MultiModelling in Fig. 2.4), each model is based on a specific meta-model, which provides a specific view on the modelled application part. Following the component concept, each MBSC can make use of the functionality provided by other MBSCs via a connector. These connectors are specified in the corresponding models and can be applied for a horizontal or a vertical connection of two MBSCs as demonstrated in Fig. 3.2. The type of connection is reflected in the deployment of MBSCs at runtime; while horizontally connected MBSCs are deployed in the same runtime node, vertically connected MBSCs should by deployed in different runtime nodes for supporting the seperation of execution states.

An example for horizontal connection is the usage of an MBSC, which defines a distinct control, in an MBSC used for a dialog specification. An example of a vertical connection is the usage of the dialog defining MBSC by an MBSC containing a specific business process.

As each MBSC defines all model elements required for its execution in its own models the specified connectors can be used for compatibility checks between the local model and the currently used model of the connected MBSC enabling to deal with distinct models resulting from the MBSC evolution as discussed in Sect. 6.7.

3.3 Model-Based Middleware

The first part of this chapter has discussed the usage of multiple models covering distinct parts of an application specification and the split-up of these models. On the other side the bundling of specific model parts in an artifact following the definition of a component has been proposed in the MBSC approach. In this section the design of a middleware targeted for the MBSC execution is presented, which is accomplished by the application and organization of the required MCC instances. This middleware is also discussed in Sect. 6.8.

Fig. 3.3 gives an overview of the components used in this middleware. Note that each of this components can be defined following an existing component model (e.g. SOFA, Fractal or OSGI). The interfaces of these components are driven by the evolution of the model interpreting middleware and are thus independent from the application evolution, which is solely defined by the corresponding models.

The number of types of MCCs is determined by the required meta-models of the executed MBSCs. Each MCC is thereby representing a plug-in in the model interpreting middleware as discussed in Sect. 6.6. This design is following the micro kernel design pattern as proposed by Buschmann et al.:

"The *Microkernel* architectural pattern applies to software systems that must be able to adapt to changing system requirements. It seperates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration." [BMR+96, p.171]

The customer-specific parts are hereby realized via the corresponding MCC types, which also act as the technical components proposed in the initial MBSC approach (see Sect.6.1).

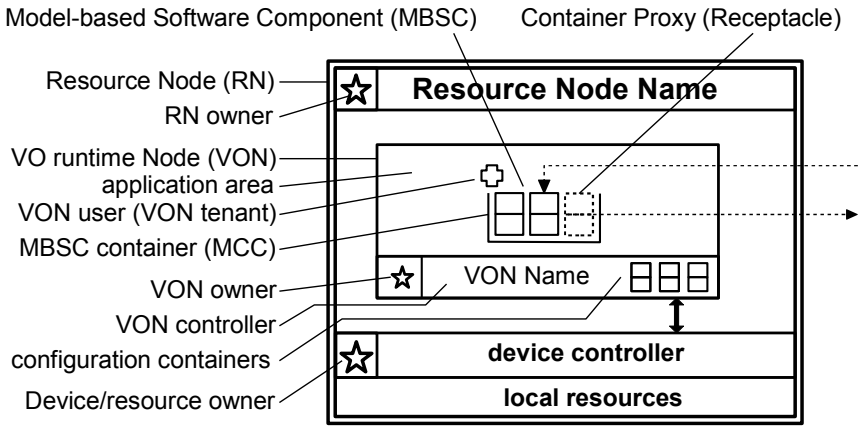


Figure 3.3: Overview of model interpreting middleware framework

The micro kernel itself is realized by two specific components, which are discussed in the following section.

3.3.1 Resource Node

A resource node (RN) is the essential part of the model-based middleware framework. It provides the ability to manage the computing and I/O resources used by the interpreted MBSCs. Each hardware device participating in the distributed system (made up by all devices running the model-based middleware) is required to execute at least one resource node. The owner of the resource node is the same as the owner of the device executing this resource node. He is responsible for the identification of the RN managed hardware components by the selection of specific methods. These methods are applied as TME statements extending the ECQL meta-model and can afterwards be used in ECQL statements of the MBSCs executed on this resource node.

As noted before the execution of an MBSC is performed by corresponding MCCs. These MCC instances are not directly executed and managed by the resource node, but are bundled in an additional component called virtual organization node discussed in the following section.

3.3.2 Virtual Organization Node

A virtual organization node (VON) is used for executing one or more MBSCs by managing the required MCC instances holding the models or providing access to MCCs running on other VONs via proxy MCC instances. One of the main differences between an RN and a VON is the ownership of the corresponding instance.

While the last section motivated the identical ownership of the hardware device and the RN, this requirement is not targeted for the ownership of a VON. Considering the applica-

tion of this model-based middleware in a distributed system following the Grid Computing approach, the deployment of partial applications specified as MBSCs on a hardware device to meet required Quality of Service constraints seems feasible. In this situation the owner of the RN can create a new VON managed by his RN while changing the owner of this created VON to the owner of the deployed MBSC. Because a VON only can access a subset of the hardware available in the RN and this subset is specified upon the creation by the owner of the RN, this approach realizes a sandbox for the executed component as available in current specifications of virtual machines (e.g. Java Virtual Machine).

A VON as suggested by its name can be a member of several virtual organizations. This enables to control the usage of the MBSCs executed by a VON by MBSCs running on another VON, as both VONs are required to belong to the same virtual organization.

As noted before the migration of MBSCs is a feasible use-case of this middleware, which is realized by the migration of a complete VON to another RN. Note that each VON is addressed by a specific Uniform Resource Name (URN), which is independent of its currently executing RN. The RN is responsible for providing a mapping between the URN address and the current address used for reaching this device.

Chapter 4

Evaluation and case study

Specific parts of the presented architecture have been implemented prototypically. The implementation was evaluated in case studies in the business domain of logistics. Some specific data of these case studies is depicted in Tab. 4.1. Two case studies targeted the design of model-based software components on mobile clients of the software WAMAS® provided by Salomon Automation GmbH, while the third case study targeted the execution of MBSCs in the client specific part of the RFID middleware You-R® OPEN, which is executed on embedded devices (e.g. RFID readers).

4.1 Supporting mobile clients in WAMAS®

The first two case studies were realized in the business domain of warehouse logistics. In this business domain a warehouse management system (WMS) is used for supporting the various key functions of a warehouse (yard control, receiving and staging (aka incoming goods), opening, counting and ticketing, internal transportation, storage, order pick and distribution, packaging, weighing and manifesting,customer returns and out-of-season product transfer and staging and manifesting) as defined by Mulcahy in [Mul93]. A traditional WMS is organized following the client-server paradigm, where the client program is executed on desktop computers, mobile handheld devices or pick-by-voice clients. Fig. 4.1 gives an overview of different mobile devices, which are supported by WAMAS®.

While a traditional WMS is operated for one specific warehouse or a particular group of warehouses, a WMS designed to support recent economic trends like supply chain management should support different stakeholders. For an analysis of these stakeholders different levels of interests are defined for the business domain of warehouse logistics:

Warehouse: People working in a warehouse are assisted by mobile devices like mobile handhelds, pick-by-voice clients and liftfork terminals. All of these devices are connected by wireless technology allowing data communication with guaranteed quality of service.

Company: Mobile handhelds are also used between two warehouses, e.g. in assisting lorry drivers delivering goods to end consumers. In this use case wireless protocols

		WMS Supplier	WAMAS	YRO
Models	Data	○	○	●
	Behavior		○	●
	User interface		○	
	TME		X	X
	Interface	X	X	X
Platform	J2ME	X		
	.NET CF		X	
	.NET MF			X
Communication	XML		X	
	JSON	X		X
	2PC	X		X

Table 4.1: Overview of case studies

for larger distances (GPRS, UMTS) are used, and security at the communication layer gets more important. Another use case of a WMS targeting this level of interest is the request of a warehouse worker to the stock information of another warehouse in the same company.

Virtual Organization: External persons supplying the warehouse are equipped with mobile phones allowing them to communicate with a WMS, e.g. for providing information about their arrival and their delivered goods. In this level communication with several WMS at the same time is an important requirement (e.g. for scheduling the deliveries at different warehouses).

Two case studies were done for use cases situated in the incoming goods stage of a warehouse. One case study was brought to completion as part of a master thesis [Kre08] and considered the stakeholders of a Virtual Organization formed by a warehouse and its external suppliers. The second case study was conducted on an industrial prototype, which consisted of a mobile client application assisting warehouse workers during the incoming goods stage.

Both case studies were based on the same domain model (containing the logistic specific data structures). The first one realized a workflow for announcing the data of delivered goods before their arrival in the warehouse. The second case study implemented a workflow of registering the delivered items in a warehouse.

4.1.1 Client for Virtual Organization

In the master thesis the concept of the EC was extended to support distributed transactions in form of a two phase commit protocol based on JSON for supporting mobile clients operated by external persons of a warehouse. The delivered goods were first registered by the supplier in his mobile application and were then selected for delivery in a specific warehouse.

(a) Datalogic Kyman (b) Datalogic Viper (c) Symbol MC9090

Figure 4.1: Some mobile devices supported by WAMAS®

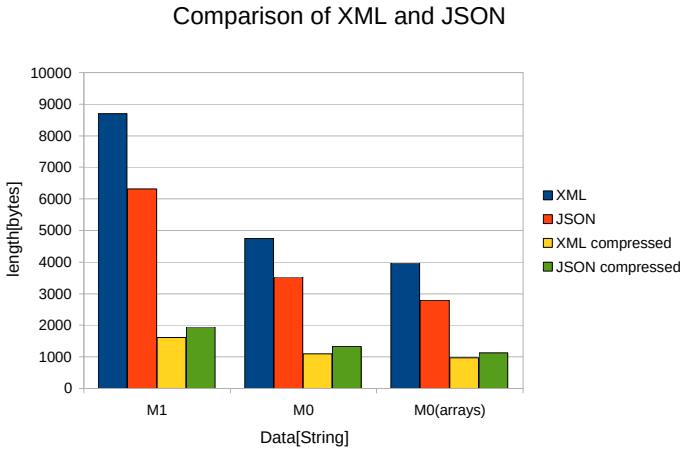


Figure 4.2: Comparison of XML and JSON encoding

The main focus of this case study was the encoding of the model-based data transmission using the JSON protocol, thus providing the basic mechanism for synchronization of MBSC runtime nodes. Although the model based interpretation of user interfaces was considered, the user interface was realized following a traditional programming approach, because of the missing MBSC basic framework at the time of this case study. Nevertheless the design of the user interface components was standardized by the application of Object-oriented design techniques in a way that would enable the transformation of the code driven specification in a model of the user interface.

A prototypical application was realized using J2ME CLDC, which is the widest supported standard for writing mobile phone applications and the Jade agent framework for enabling communication with the mobile phone. A dialog for entering article related information is depicted in Fig. 4.4a. Note that the article in these panel could be provided by one MBSC, which could be loaded from the selected WMS, thus allowing the input of Warehouse specific article information, while the rest of the dialog does not need to be changed.

The usage of JSON for data encoding proved useful, which is demonstrated by Fig. 4.2.

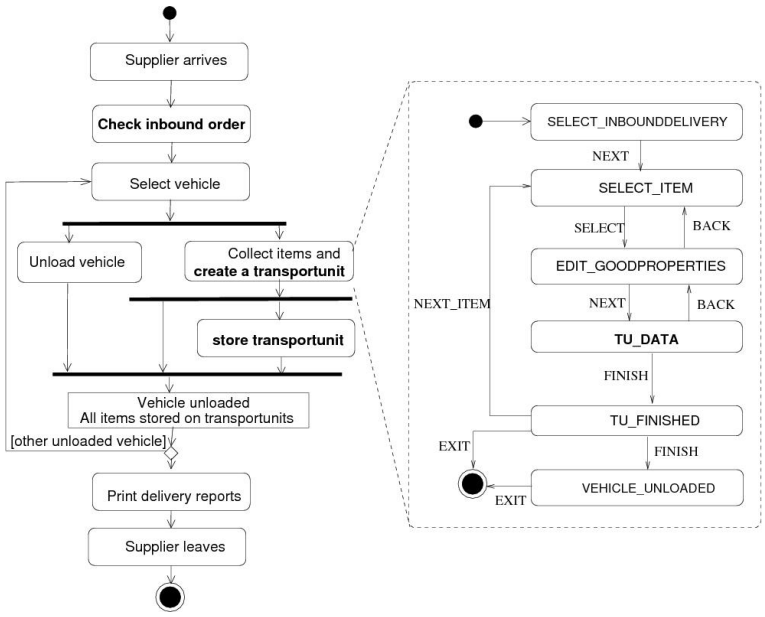


Figure 4.3: Realized incoming goods process

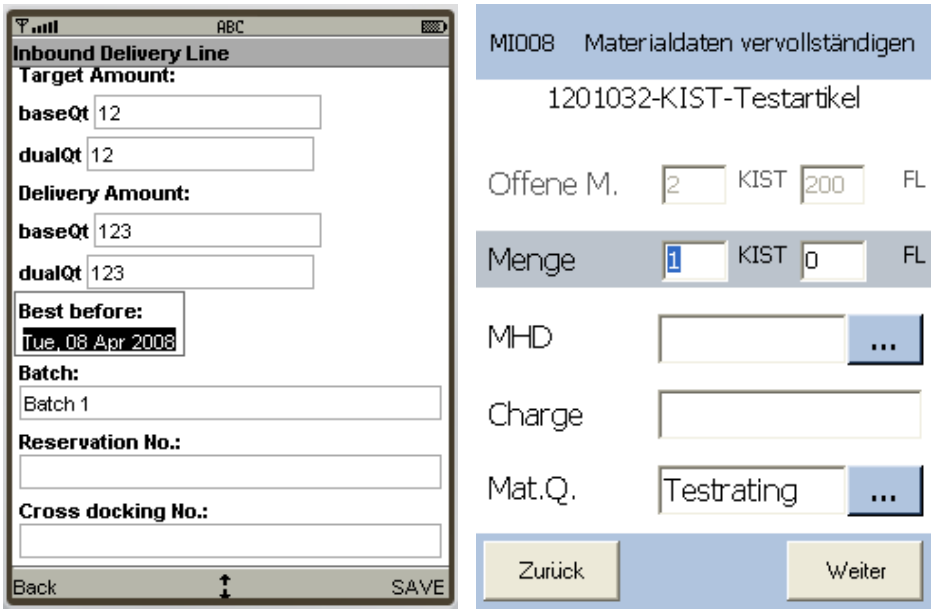
Note that the best solution consisting of compressed XML files requires an additional conversion step, which is a time and energy consuming task. The JSON encoding of the model data was refined in the third case study, which will be discussed in Sect. 4.2.

4.1.2 Client for incoming goods stage

The second case study was accomplished on a mobile client supporting warehouse workers in an incoming goods process for a logistics provider.

In this case study the behavioral description of the warehouse tasks and the user interface activities using hierarchical state machine models was tested. The client application was realized using the .NET Compact framework and applied the behavioral model and the data model for generative purposes, while the user interface model was specified in source code based on a first version of a user interface plug-in. This plug-in provided a specification of common controls used fro the mobile client and was also refined and implemented in a second prototype based on J2ME CDC. Instead of JSON the data representation of this case study was based on XML because of the used webservice technology for communication purposes between the .NET-based client and the Java-based server.

Fig. 4.4b depicts a dialog used by the warehouse worker for the specification of related information of an incoming item. Note that the same information is captured by this dialog and the dialog presented in the previous case study, which is based on a compatible data model.



(a) external supplier (b) warehouse worker

Figure 4.4: Dialog for specifying article related information

4.2 Moving functionality from You-R® OPEN to RFID readers

The third case study was performed on the shared usage of reader embedded functionality in an RFID middleware [Lei10]. Based on the results of the two first case studies and the proposed MBSC architecture a MCC based runtime node architecture was implemented using the .NET Microframework [Kuh08]. While the case study in WAMAS featured code generation or the application of language specific frameworks (e.g. webservice support provided by .NET) to apply the MBSC models, this project featured a full implementation of the MCC architecture, enabling the interpretation of data and behavioral models at runtime. Furthermore the resource node (discussed in Sect. 3.3.1) and the virtual node component (discussed in Sect. 3.3.2) of the model-based middleware framework were implemented.

Based on this implementation a scenario targeting the support of shared usage of an RFID reader in a building access control system has been evaluated in [Lei10]. Four organizations are involved in this scenario, each running at least one virtual organization node on the shared RFID reader as depicted in Fig. 4.5. The owning organization of the RFID device is the facility management of the building, which is also operating two VONs on the device. The VON FacilityMgmt N1 is executing an MBSC to split up the data provided by detected RFID tags on this reader. This MBSC is used by two MBSCs executed in the corresponding VONs of the departments located in this building. Each MBSC implements an authentication mechanism to check if the holder of the detected RFID tag is granted

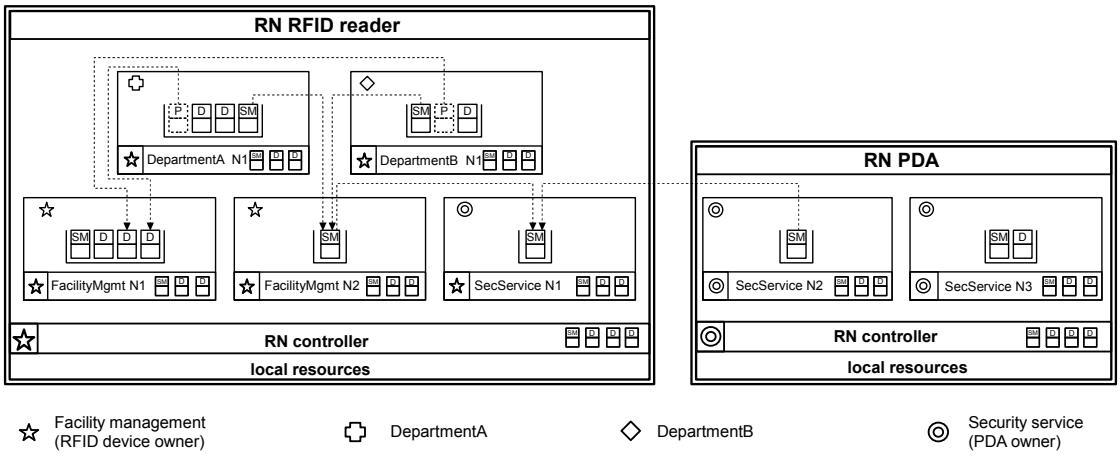


Figure 4.5: Scenario for RFID case study [Lei10]

access to the building. The second VON owned by the facility management is used for separating the concerns between the departments and the security service, which is also running an MBSC on the reader. This MBSC is used for locking and unlocking the door managed by this RFID reader.

Additionally a mobile device owned by the security organization was included in this scenario. Two MBSCs were executed on this device for controlling the door and displaying a log message for each user trying to access and leave the building. Several virtual organizations are created in this scenario:

- One VO is defined by the MBSC FacilityMgmt N1, DepartmentA N1 and FacilityMgmt N2.
- Another VO is defined by the MBSC FacilityMgmt N1, DepartmentB N1 and FacilityMgmt N2
- The third VO is containing the MBSC FacilityMgmt N2, SecService N1, SecService N2 and SecService N3.

This scenario demonstrated the realization of different VOs on a shared embedded device. Also the migration of MBSCs to other devices can be studied in this scenario, by moving the MBSCs owned by the departments to other resource nodes. Leitner also presented measurements for the actions required for migrating or updating an MBSC as visualized in Tab. 4.2

The storage requirements for the applications running on the reader are depicted in Fig. 4.6. While the middleware part is a static part regarding the application functionality, the size of the interpreter is determined by the different views provided by the executed MBSCs. In this scenario two interpreters where applied to deal with the data view and the behavioral view of the executed MBSCs. The device specific functionality is also affected by the different extensions in the ECQL statements of the MBSC to access device specific

CPU speed [Mhz]	.NET platform	Time [s]
200	Micro	10
600	Compact	2
1400	Full	1

Table 4.2: Transferring models to a VON [Lei10]

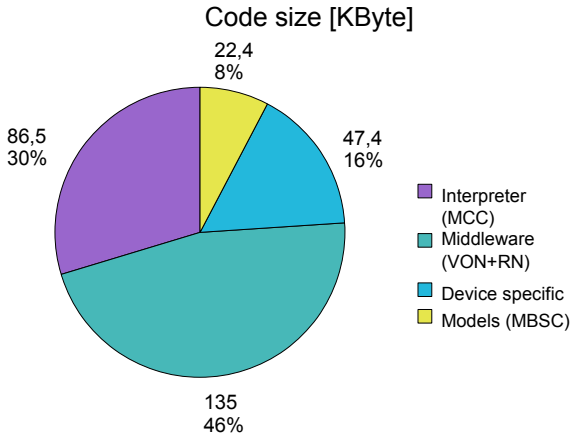


Figure 4.6: Code size of implemented framework and models used in scenario

functionality. The size of the models is determined by the number of different MBSCs running on this device. In this case study five MBSCs were executed on the RFID reader and two MBSCs were executed on the PDA owned by the security service.

The case studies have demonstrated the feasibility for using models to define application parts in a structured and reusable way. The results provided by Kremser and Leitner demonstrated the performance improvements for the communication by the application of JSON and the minimal storage requirements for the models compared to the framework. On the other side the results presented by Leitner on the increased memory requirements of the running application pointed out the drawbacks of the multi-layered model hierarchy. This disadvantages could be reduced by the application of programming languages with a lower abstraction for coding the framework (like C) and the usage of more static structures for managing the model elements.

Chapter 5

Conclusion

A runtime system for executing model-based software components has been presented in this work, which targets to increase the support of distributed application development for mobile and embedded devices. The proposed runtime system is targeted for distributed systems made up of mobile and embedded devices, which are indicated by constraints on the devices resources (e.g. CPU cycles, memory, available power) and unstable network connections. A three-layered architecture has been introduced consisting of application layer, middleware layer and resource layer. This architecture has been used for illustrating the heterogeneity of the artifacts contained in each layer and the different roles of people managing and using these artifacts.

5.1 Overview of proposed framework

Related work has been presented on the usage of model-based techniques for separating the application domain specific concepts from the target platform specific code. Also currently proposed software component models for mobile and embedded devices have been examined. As a result of discussing the related approaches a missing modularization of the software models used for defining the application domain was recognized. Furthermore the limited usage of software models in current component models was noted; the approach of feature modeling for selecting the relevant parts of the software component model was found to be problematic regarding the resource constraints of mobile and embedded devices.

The proposed runtime system is situated in the middleware layer and is made up of several components. A model-based component container (MCC) is used for interpreting a given model, which represents a specific view on a model-based software component(MBSC). For accessing model elements managed by other MCCs a model-based action and query language has been defined. All MCCs holding the models defined for an MBSC are managed by a virtual organization node (VON). An application made up of MBSCs is therefore executed by several VONs lying in the same virtual organization, which is an organizational concept proposed in the field of grid computing supporting a "[...] dynamic group of users with a common goal coming together for a specific, short-lived collaborative venture

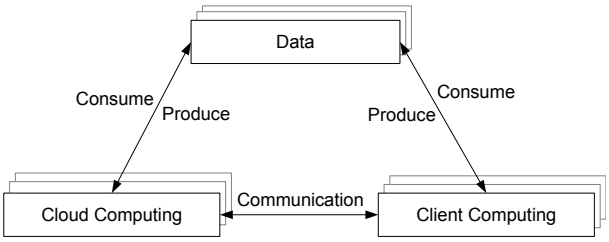


Figure 5.1: Triangle of next generation Internet Computing [FZRL08]

[...]"[BJK09, p41]. All VONs of a device are hold by a resource node (RN) component, which manages the access to the hardware resources by assigning designated hardware resources to specific VON instances. This concept provides a separation between the owner of the hardware and the owner of a deployed MBSC, enabling a trusted management of the used hardware resources by an MBSC.

The proposed framework has been evaluated in several case studies featuring two software products in the business domain of logistics (a warehouse management system and an RFID middleware), which are currently handling the used mobile and embedded devices as following a thin-client approach. This solution requires a good network connection while the resources of the devices are not fully accessible. The case studies have outlined the support of the proposed framework for enabling application specific functionality on the mobile and embedded devices while respecting the constraints of the distributed system and its members. Examples of such functionality are the collection of goods with a mobile industrial handheld without a network connection or buffering read RFID tags locally on an RFID reader in case of network problems.

As a consequence this architecture can be used to manage the heterogeneity on each layer, by concentrating the usage of general purpose programming languages and corresponding component frameworks in the middleware layer, while the solution of application specific problems is specified in different models, each one supporting the best fitted modeling technique for the problem managed by the specified view. A separation of concerns is introduced by the microkernel based architecture consisting of resource node and virtual organization node components and of model view specific model-based component containers acting as plug-ins to the runtime system. Fig. 5.1 depicts this separation of concerns as proposed by Foster et al. in their analysis of current cloud computing approaches [FZRL08]. By enabling the dynamic placements of executed MBSCs (containing the data of the application) on VONs running on other devices (the cloud computing aspect) the user has full control of its application within the targeted virtual organization (the client computing aspect).

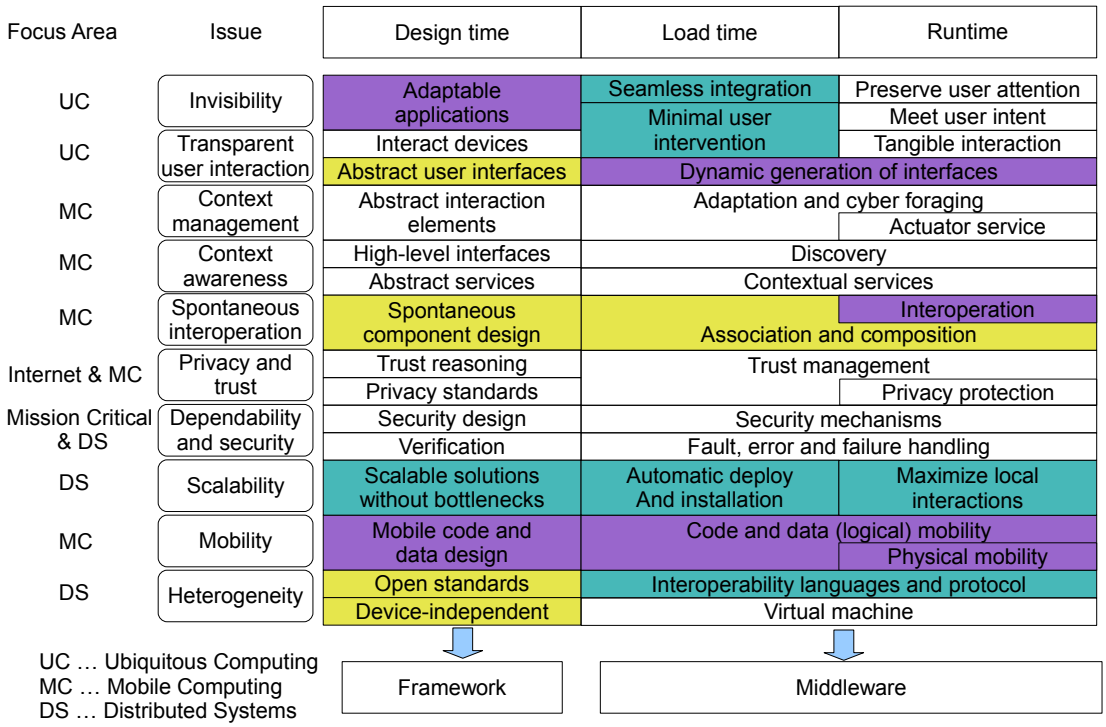


Figure 5.2: Issues in ubiquitous computing systems based on [dCYG08]

5.2 Future work

The proposed framework is targeting the characteristic challenges for distributed pervasive systems as noted by Tanenbaum and van Steen [TvS06]. The pervasive computing paradigm has emerged out of the mobile computing paradigm as noted by Saha and Mukherjee, stating that "the 'anytime anywhere' goal of mobile computing is essentially a reactive approach to information access, but it prepares the way for pervasive computing's proactive 'all the time everywhere' goal." [SM03, p.26]

But again this paradigm and the corresponding distributed system can only be seen as an intermediate step towards the vision of ubiquitous computing as postulated by Weiser [Wei95]. This fact is also underpinned by the issues reported for ubiquitous computing by Costa et al. [dCYG08], which are illustrated in Fig. 5.2.

According to their categorization distinguished solutions are provided in different lifetime phases of an ubiquitous system. While framework support is required for tackling issues at the design time, a middleware should provide the corresponding methods at the load time and runtime of the ubiquitous system.

While the issue of heterogeneity is targeted by the application of software models, the issue of mobility is solved by the various types of MCCs and their platform specific implementation. The scalability is managed by the resource nodes and the virtual organization

nodes of the architecture, which are also responsible to manage the privacy & trust and the dependability & security issues.

This last two issues require further research in the integration of model-based techniques for analyzing the behavior of the application in the framework, to detect a fraud behavior in early stages.

Another field of future work is the realization of additional models to foster the invisibility of an application as well as to consider the further abstraction of the user interface. Also solutions for supporting the deployment and execution of MBSCs on VONs and RNs depending on the current context of this framework components should be considered.

Chapter 6

Publications

This chapter provides the publications written during this thesis ordered after the significance to the contributions as discussed in Sect. 2.3 and depicted in the layered architecture in Fig. 3.1.

The concept of Model-Based Software Components is described in Sect. 6.1, which has been presented at the 16th IEEE International Conference and Workshops on Engineering of Computer-Based Systems in San Francisco in 2009. The usage of several models containing the functional specification of a component as well as the connection of several components with specific model connectors is discussed in this publication. The runtime architecture described in Publication 6.5 and 6.8 is represented by the technical part of the Model-Based Software Component. This paper also presented the results of the case study on the industrial prototype of a mobile application supporting warehouse workers in the incoming goods stage, which has been conducted with the company Salomon Automation GmbH.

A domain specific language for accessing the other models of the specified MBSC and its connected MBSCs is discussed in Sect. 6.2. This paper has been presented in April 2010 at the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems in Oxford and is based on a scenario developed in the case study of the RFID middleware.

The second WMS specific case study discussing the usage of internal and external views on models of an MBSC has been presented in Sect. 6.3 at the IASTED Conference on Parallel and Distributed Computing and Networks in 2008. This publication focused on the sharing of data models and presented the case study of a supplier working with a warehouse management system.

As data models for desktop and server systems where the initial application domain of the EntityContainer, the concept of Transient Model Extensions has been also demonstrated with this kind of models. The results of applying this concept on different tiers of an information system are discussed in Sect. 6.4 and Sect. 6.5. They have encouraged the usage of this concept for connecting the different models of an MBSC at runtime.

This connection has been enabled by the application of the TME concept on the layered EntityContainer architecture at runtime, which resulted in a runtime plug-in system for the MBSC concept as presented in Sect. 6.6.

For the connection of various nodes (each executing different MBSCs) of this runtime plug-in system, the concept of component connectors based on model compatibility as discussed in Sect. 6.7 is applied. This paper has been presented at the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA).

Finally Sect. 6.8 contains the publication at the 15th IEEE International Conference on Engineering of Complex Computer Systems, which discussed the results of the RFID middleware case study focusing on the shared usage of resources and the separation of MBSC execution nodes and hardware resource management nodes.

Interpreting Model-Based Components for Information Systems

Michael Thonhauser, Christian Kreiner, Martin Schmid
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
michael.thonhauser@TUGraz.at

Abstract

To foster the reuse of software artifacts various approaches like Component Based Software Engineering or Model-Driven Software Development have been proposed. These approaches support a developer in generating and implementing platform specific software artifacts, which can be executed on the chosen runtime architecture. To facilitate portability of these artifacts to other runtime architectures it is important to model various aspects of the artifact (i.e. user interface, behavior, data) in a platform independent way. While this abstraction helps to reduce complexity of the problem, choosing the right granularity of methods provided by this artifact is another important issue for enhancing software quality.

Considering these aspects a model-based development approach is presented, which is based on the interpretation of several model views – like state machine and class diagrams being provided by a model-based software component. Additionally the integration of components build with this approach in the design of an information system is discussed. The proposed architecture is evaluated by an implementation in the software application domain of logistics.

1. Introduction

Dealing with business processes distributed over a large landscape of existing and new heterogeneous systems that are under the control of different owners is a main goal of the Service Oriented Architecture paradigm [6]. SOAs are build on services providing well defined interfaces, thus hiding technical details of a service implementation. They can be loosely coupled to reduce dependencies between the different systems and can be composed to realize complex requirements.

Loose coupling is also one of the main motivations for the application of Event-Based Systems [9]. Application scenarios of systems, which are designed following this approach, contain distributed event publishers and subscribers, being loosely coupled through events. This paradigm is also well suited for mobile devices, like cell phones or personal digital assistants, acting as participants of such systems to deal with the constraints of the mobile connection.

Different approaches for solving the constraints on mobile devices (like energy management, context awareness or user interfaces) have lead to an increasing number of different hardware platforms and operating systems, which constitute a heterogeneous execution environment.

To deal with these heterogeneous runtime environments while enabling a better support for both types of loose coupling, Model Driven Software Development (MDSD) techniques [15] can be applied. The development of such models allows the reduction of platform dependencies and the usage of modeling tools (like Unified Modeling Language editors or editors for Domain Specific Language). In most applied MDSD approaches these models are used at development / build time for generation of implementation skeletons for services or state machines. The problem with this approach is the requirement to run the generator after each modification of the model and the reduction of information, because each generated artifact only contains a subset of the information contained in the original model.

Another possible MDSD technique discussed in [15] is the interpretation of models, which allows a late binding and dynamic reconfiguration of the model based software artifacts.

Based on this approach a framework is introduced in this paper, which supports the execution of model based software artifacts in a distributed business process realized on mobile devices. These software artifacts are defined as components containing a model of the data, the behavior and the user interface. Actions provided by the realizing component are implemented as platform specific scripts, which are used by the event based runtime platform. Composition of the components is based on compatibility of the contained model views, thus increasing the stability of the interfaces between different components. The support of this approach for reduction of platform specific code and increased reusablity is discussed by an example in the business domain of logistics.

2. Related work

Several software engineering approaches have been proposed to foster the reuse of software artifacts. While Component Based Software Engineering (CBSE) has been focused on the definition of deployable software artifacts, MDSD has been introduced to allow a specification of these artifacts in a platform independent way.

2.1. Software components

Several definitions of a software component exist. According to a basic definition given in [16], a software component is a "unit of execution with well defined interfaces" .

Handling of software components is specified in a software component model. A taxonomy of various software component models (such as Enterprise Java Beans or COM) is given in [8].

Another component model, which is built on top of a Java Virtual Machine, is proposed by the OSGI Alliance. Components following this model are developed as bundles providing services and extension points, which can be used by other bundles being plugged into the OSGI runtime environment. This component model serves as basis of the Eclipse Integrated Development Environment, allowing for a broad coverage of useful features required in a software development process through associated projects.

Service Component Architecture (SCA) [5] is a component standard, intended for assembling heterogeneous service implementations in a SOA. Components following the SOAC approach [14] are categorized in four different categories (service components, business components, adapter components, utility components), but like SCA are missing the support for specification of their executional aspects based on a platform independent model.

A component model proposed in the field of Grid Computing are Gridlets [19], which can be seen as a chunk of data associated with the operations to be performed on the data. These operations are provided by corresponding binaries.

The KobrA component approach has been introduced in the field of software product line development [1]. In this approach several models are used for the description of the component in the specification and realization phase. Although this approach also relies on class diagrams and state chart diagrams in the specification phase it is missing a description of the user interface and also does not provide support for interpreting these models at runtime.

2.2. Model Driven Development

The 4+1 view model of software architectures at system level has been developed by Kruchten [7]. This model consists of different views on the described software architecture targeting different aspects and responsibilities:

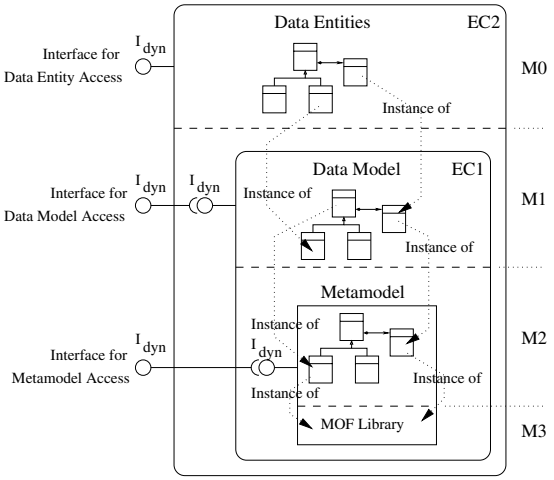


Figure 1. Four-level metamodel hierarchy implemented in the Entity Container [12]

Logical View:	containing end user functionality
Development View:	for programmers and software managers
Process View:	for integrators describing performance and scalability
Physical View:	for system engineers concentrating on the topology and used communication mechanisms
Scenarios:	illustrating the architecture with selected use cases.

Each of these views can be specified using different notations. Such a notation is for instance available in the Unified Modeling Language (UML) [11], which is maintained by the Object Management Group (OMG). Two kinds of models are distinguished in the UML, allowing the structural and behavioral specification of software. Each kind contains several diagrams for visualizing specific views of the modeled software. Every diagram supports different phases of a software development process.

Class diagrams are applied for the description of the structural software parts and are one of the most commonly used views of a software model.

Each model can be used as a metamodel for a new model, which is defined by the four-level metamodel hierarchy of the OMG [10]. In this hierarchy, layer M3 contains the base model for all modeling languages, the Meta Object Facility (MOF). UML language elements contained in layer M2 are instances of MOF elements. A user model is defined in the layer M1 with its elements being instances of UML elements. Finally, M0 contains the runtime elements, being instances of M1 model elements. Beside UML also other approaches built on this metamodel hierarchy, such as the

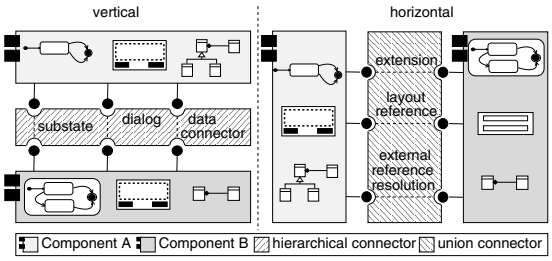


Figure 2. Functional MBSC composition

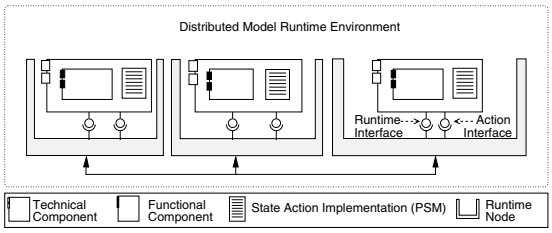


Figure 3. Functional vs. technical component

Eclipse Modeling Framework (EMF) [2].

Various approaches for modeling systems have been proposed, such as the statemate approach [3], which relies on a combination of activity charts and state charts to model reactive systems.

Another approach making use of state charts in the construction of user interfaces is described in [4].

2.3. Entity Container

The Entity Container (EC) [12] has been proposed as an object-oriented, model-driven data persistency cache. It is configured by a data model and can load and store data with several different persistency mechanisms (such as XML files or relational databases) via connected backingstores. Fig. 1 depicts the realization of the UML metamodel hierarchy by recursive usage of several EC instances, which are configured respective loaded with the M3 – M0 models of the metamodel hierarchy.

Based on the EC the concept of model-typed component interfaces [13] has been developed, which defines compatibility rules for connecting components, which are based on different, but compatible data models.

Also the concept of Transient Model Extensions (TME) has been introduced in [18], allowing the temporary extension of the metamodel used as the basis for the EC. This extension can be combined with platform specific code snippets, which are called as handlers or observer callbacks if a TME defined element is manipulated in the EC.

3. Model-based software components

The proposed approach of Model Based Software Components (MBSC) distinguishes two views on a software component, i.e. each MBSC is made up of a functional part and a technical part.

The **functional part** of a MBSC consists of the provided models for behavior, data and user interface. Composition of components is done in the functional view by combining the corresponding models of several MBSCs through the Distributed Model Based Runtime Environment. Two types of MBSC composition are depicted in Fig. 2 , i.e. a vertical composition by using hierarchical connectors and a horizontal composition by applying union connectors. The mechanism used for composition depends on the composed model.

Statemachines model the behavior of a software component. Hierarchical composition of statemachines is performed with *substates*. One example of a hierarchical composition is the combination of one MBSC containing the model of a business process, with several MBSCs, each defining one dialog to be shown in one distinct state of the process.

Parallel substates are possible, to connect more than one MBSC to a master component. An example for the composition using parallel substates is the combination of a dialog component with several MBSCs containing panels to be shown in the dialog.

Horizontal composition of statemachines is triggered by special states, which allow the extension of the current statemachine with additional states and transitions.

For horizontal composition in the **user interface** model a layout reference can be defined, which is filled with the user interface of a composed MBSC. Vertical composition of user interfaces leads to a new dialog.

Vertical composition of **data models** requires them to be compatible with the rules defined in [13] by using the concept of data connectors presented in [17]. Horizontal composition at the data level is realized by resolving external model elements and connecting their references with the current data model.

The **technical part** of a MBSC makes use of existing component models (like EJB, .NET, CCM). According to Fig. 3 one functional part is contained in a technical part of an MBSC. For each MBSC the persistent state of the various MBSC models (statemachines, data, user interface) is hosted by the technical component, which is also used for supporting component migration. Additionally state action methods are provided, which are implemented in the programming language supported by the component model.

The persisted models, the data, state action methods and TME configurations for extending the metamodel of the runtime are provided through an interface required by a

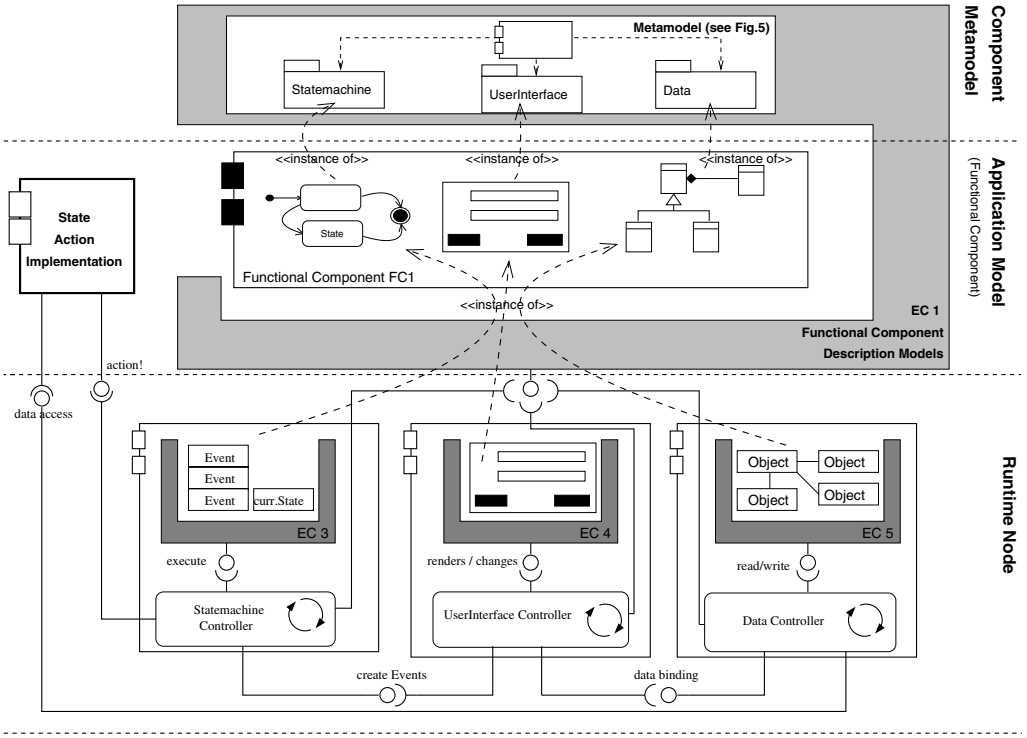


Figure 4. Model Based Runtime Node

runtime node component of the Distributed Model Runtime Environment.

Note that changes in the functional view (e.g. changing business processes leading to changes in the behavioral definition or additional data fields resulting in a changed data model view) do not require changes in the technical interface, therefore increasing the stability of a technical component.

3.1. Component Runtime

The MBSC runtime environment consists of distributed nodes, as depicted in Fig. 3. Each runtime node is managed by the container of the chosen component platform and is usable as an execution platform by several MBSCs. Fig. 4 depicts a structural overview of the components contained in one runtime node and their relationships and location depending on the four level metamodel hierarchy.

The *component metamodel* (corresponding $M2$ in the four level metamodel hierarchy) is the common basis for all nodes of the MBSC runtime environment. It defines the elements available in the three model views (statemachine, data, user interface) of a MBSC, using the elements defined in MOF ($M3$).

The *application model* level (corresponding to $M1$) is defined by the MBSC designer. Note that the elements contained in the functional component $FC1$ are instances of the component metamodel.

The *runtime node* level of the MBSC runtime consists of the ECs managing the corresponding data objects, which are created by the interaction of the different controllers. While data objects (managed by $EC5$) are direct instances of the data model, $EC4$ primarily contains one instance for each user interface model element defined in the user interface model of $FC1$, and $EC3$ contains the management data of the current statemachine controller (e.g. current state and a queue of waiting events). Each EC in the runtime node level contains platform independent data items, which are interpreted in a (component) platform specific way by the corresponding controllers.

For example the user interface is provided in a platform independent way by $EC4$ and is rendered by the user interface controller using a supported widget library (e.g. SWT, Java Swing or .NET). The platform independent representation of the user interface allows mappings of the user interface to other interface techniques like speech communication libraries or EAI integration interfaces.

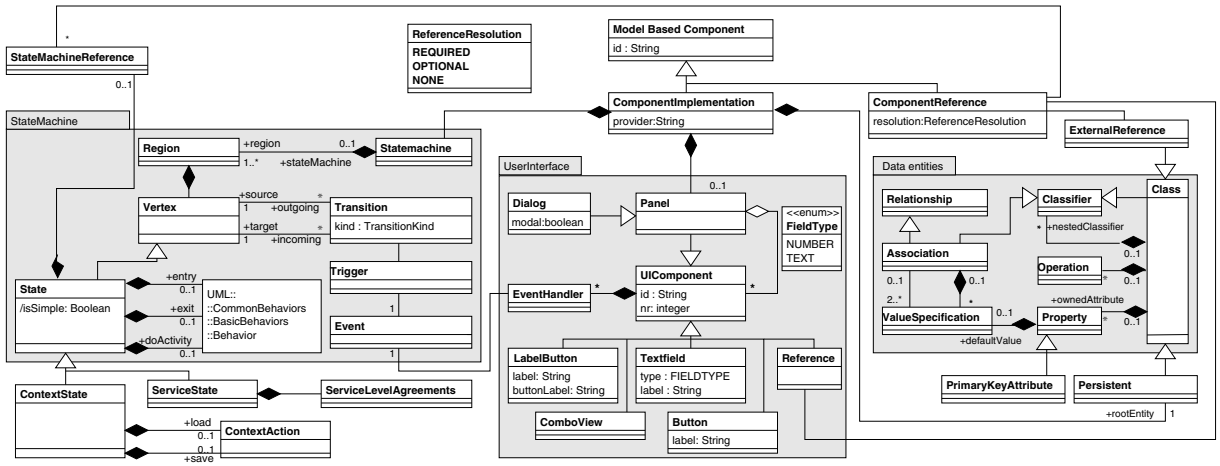


Figure 5. Metamodel of a model-based software component

3.2. Runtime metamodel

Like mentioned in Fig. 4 the definition of the elements available in the models of a MBSC is provided by a component metamodel. A simplified metamodel is depicted in Fig. 5. The metamodel is divided in three parts, each defining the elements used in one corresponding view at the application model level. While the elements contained in the data and the statemachine package are based on the corresponding elements defined in the UML metamodel for classes and behavioral statemachines, the elements of the user interface package are based on common widgets available in standard toolkits for graphical user interfaces.

Some elements in the metamodel (like subclasses of *State* or subclasses of *UIComponent*) are specifically interpreted by the corresponding controllers in the runtime node. Although the evolution of the component runtime also leads to additional elements in the metamodel, situations can occur, where the provided metamodel is not sufficient for the currently developed MBSC.

In this case the MBSC developer can define additional elements in the metamodel by providing a TME [18] with the additional element and platform specific code snippets, which can be used by the runtime node controller to handle corresponding model elements in $FC1$. Providing a TME with platform specific handlers reduces the supported runtime node platforms , but this compromise is needed to make this approach practicable. Also more than one handler can be provided for a TME, therefore multiple platforms can be supported by a TME based MBSC.

The supported target runtime node platforms are also limited by the implemented action methods of a MBSC, which are defined in a platform specific way too. But like in TME multiple technical components can be provided for one functional MBSC. Because the data structures, the behavior and the user interface of a MBSC are defined in a model, the corresponding code for handling the actions should consist of methods with a few lines of code and a sub-scope of the combined data model. Therefore porting these actions to other target runtime node platforms should be easy.

The usage of the UML metamodel allows the application of UML tools for the creation of a MBSC, integrating the additionally defined elements through UML profiles. Creation of the user interface can be supported by a graphical editor, which is based on the defined user interface model.

In each definition of a view in the metamodel a specific reference class is defined, allowing the declaration of possible compositions of the current MBSC implementation with other MBSCs. These references are used by the runtime node while loading and executing a given MBSC. Each reference is associated with at least one ComponentReference, which contains an attribute for providing the preferred resolution strategy. If the reference resolution attribute has the value REQUIRED, the referenced MBSC must be available for the given runtime node to successfully execute the current MBSC. In case the reference resolution attribute has the value OPTIONAL, the referenced MBSC is used if available. If the referenced MBSC is missing and the reference resolution attribute has the value NONE, running the MBSC is possible, as long as the referenced MBSC is not needed.

4. Example

For demonstration purposes a business process for an incoming goods (IG) stage of a warehouse management system is introduced on the left side of Fig. 6.

For further discussion we are looking in the activity of *collecting items and* ***creating transport units*** for the

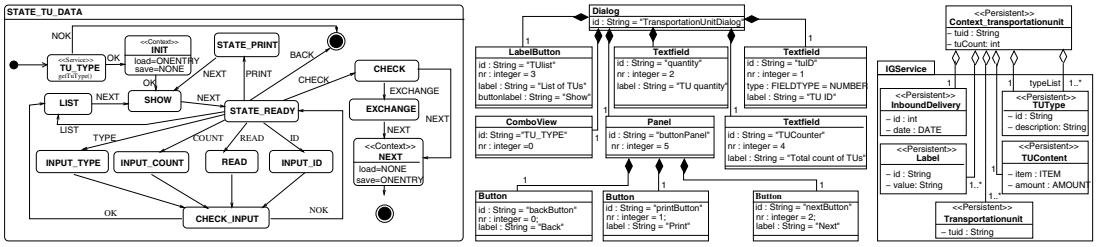


Figure 7. TU_DATA Functional Component model views

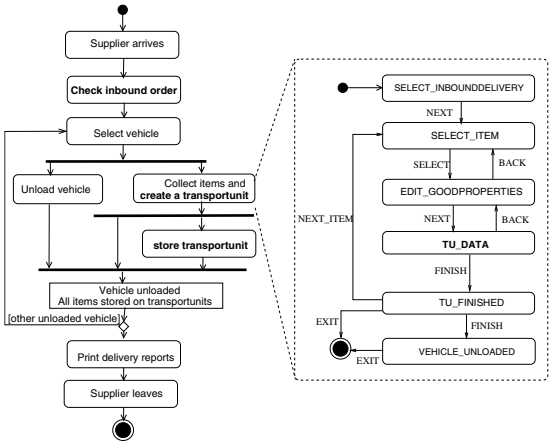


Figure 6. Example: IG business process

warehouse management system, which is performed by a warehouse worker equipped with a mobile device. This activity is implemented by a state machine shown on the right side of Fig. 6. This state machine is defined as a behavioral view of one MBSC; events, which are associated with the corresponding trigger (according to the metamodel in Fig. 5), are displayed as label of a transition. Therefore the same label can appear at different transitions of the state machine. Note however that it is not allowed for one state having two outgoing transitions with the same event.

Each step of the activity on the right side of Fig. 6 is handled using different dialogs displayed in each step of the state machine.

4.1. TU_Data component

To demonstrate the structure of one MBSC containing a user interface, the MBSC connected vertically to the state TU_DATA of the state machine in Fig. 6 is examined.

This MBSC embodies the state machine defined in the left part of Fig. 7, which makes use of the same notation as described above. This state machine contains two exit states firing events for the superior state machine, i.e. the exit state associated with state NEXT fires a FINISH event, which makes TU_FINISHED the new active state in the state machine of Fig. 6.

The state chart also contains one service state, which is used to retrieve the list of transport unit types, and two context states, containing the configuration to synchronize the data of this component with the hierarchical higher MBSC.

The data model describing the data structures available to the implementation of each of the component's state actions is presented in the right part of Fig. 7. This model contains the root entity *Context_transportationunit*, which is the central data class for this component. This class has associations to various data classes, which are instantiated during state actions of the component or by copying them from the context of the vertically connected MBSC.

The model for the user interface is depicted in the middle of Fig. 7. The class *Dialog* is the main class of this model and is associated with different controls. Each control has a unique identifier, which enables methods of this component to reference the corresponding control. The attribute 'nr' is used for defining the order of the controls in the dialog. A dialog can contain various panels, which can contain controls being layouted inside the panel (as shown in the metamodel in Fig. 5).

4.2. Implementation

The business process presented in the previous section has been implemented in an industrial prototype in the business domain of logistics. The developed MBSCs have been stored in one technical component, which has been created as a .NET assembly based on the .NET Compact Framework for mobile handhelds. Initially two business activities in the incoming goods process defined in Fig. 6 (*check inboundorder* and *create transportunits*) have been developed. The number of MBSC used in each activity is presented in Table 1.

The customized *create transportunit* activity was created by removing MBSCs, which were not used because of changed requirements in the IG process of the customer. Removing the MBSCs from the initial business activity was

Table 1. Implemented activities

		LOC[%]				
Activity	**# MBSC**	**Statemachine**	**Data**	**UI**	**UI_Logic**	**Logic**
Check inboundorder	6	9.36	0.00	68.21	11.23	11.20
Create transportunits	23	16.60	9.34	16.26	31.99	25.82
Create transportunits (customized)	13	16.28	9.40	14.47	34.17	25.68
Store transportunit	2	15.89	6.59	14.59	40.00	22.92

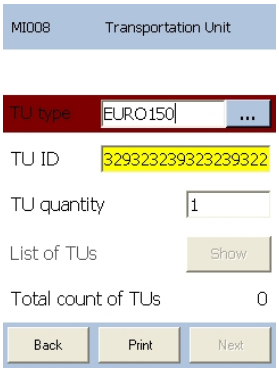


Figure 8. "Transportationunit" dialog

eased by the facts of hierarchic composition and the self containment of each MBSC.

The *store transportunit* activity has been developed to support forklift drivers, who store the transportunit in a warehouse rack. This activity contained one MBSC for the process and one MBSC containing a simple dialog for selecting the transportunit and entering the warehouse rack number.

The proposed model environment runtime architecture has been partially implemented for the statemachine and the data model. The controllers were implemented in additional .NET assemblies, while the corresponding models were realized as classes containing the structural information. The user interface has been implemented with Windows Forms using a customized controls library.

Fig. 8 presents a screen-shot of the TU_DATA dialog, which is described by the user interface model in the middle of Fig. 7.

The approach has proved to be useful, because of the split up of the process in several MBSCs with their fixed internal structure and their well defined behavior through statemachines. Table 1 depicts the percentual distribution of the lines of code (LOC) used for implementing each activity with MBSCs. The fact, that no line has been implemented for the data part of the *check inboundorder* activity can be explained by the reason, that this activity has been implemented as a visual prototype (only displaying the dialogs used in the process in their correct order). This is also the reason for the high number of code-lines used in the user interface part.

Table 2. Implementation of TU_DATA MBSC

Component part	**LOC [%]**
Statemachine	15.29
Data	8.04
Interfacer	18.24
User interface Logic	34.71
Business Logic	23.73

The platform specific source code containing the state actions is represented by the logic column, containing the code-lines used for realizing the business-logic of a component and the state action methods. Note that according to the numbers presented in Table 1 about 25% of a component need to be realized in a platform specific way, while the rest can be specified by the functional model of a MBSC.

The high percentage of LOCs for the UI_Logic column is the result of additional code, needed to handle the access to user interface elements from other threads than the user interface thread. This number could be reduced by implementing the corresponding mechanisms in the user interface controller of the runtime node.

Table 2 lists the corresponding distribution of the LOCs for the implementation of the TU_DATA component introduced in the previous section.

5. Conclusion

In this paper an approach for model-based definition of components has been presented. This approach makes use of different model views for definition of the data, the behavior and the user interface of the modeled component. For an information system implementation in the business domain of logistics, such a model proved sufficient to fully specify all relevant aspects of a component – in a platform independent way. The content of the views is based on a defined metamodel, which is based on the UML metamodel for state machines and class diagrams.

Additionally, a runtime architecture has been defined, which contains a platform specific implementation of the component metamodel. This implementation is used for creating platform specific objects at runtime from the platform independent model of this component.

Parts of the runtime have been implemented in an industrial prototype in the business domain of logistics and

have proved their support in the iterative development of clients for information systems, by enabling better reuse of the defined dialogs and business processes. A customized process for a customer could be provided by only changing the behavioral view of the process component, while not changing the used dialogs.

The next steps planned include the integration of layout models already used for desktop applications and more research about the enforcement and integration of service level agreements in the runtime architecture.

References

[1] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wuest, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[2] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Prentice Hall International, August 2003.

[3] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill Companies, October 1998.

[4] I. Horrocks. *Constructing the User Interface with Statecharts*. Addison Wesley, November 1998.

[5] Open SOA initiative. Service component architecture specifications, August 2008. http://www.osoa.org.

[6] N. M. Josuttis. *SOA in Practice, The Art of Distributed System Design*. O'Reilly, 2007.

[7] P. Kruchten. Architectural blueprints - the '4+1' view model of software architecture. *IEEE Software*, 6(12):42–50, November 1995.

[8] K. Lau and Z. Wang. A taxonomy of software component models. *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 88–95, 30 Aug.-3 Sept. 2005.

[9] G. Muehl, L. Fiege, and P. R. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[10] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, November 2007.

[11] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, November 2007.

[12] G. Schmoelzer, S. Mitterdorfer, C. Kreiner, J. Faschingbauer, Z. Kovács, E. Teiniker, and R. Weiss. The entity container - an object-oriented and model-driven persistency cache. In *HICSS*. IEEE Computer Society, 2005.

[13] G. Schmölzer, E. Teiniker, and C. Kreiner. Model-typed component interfaces. *Journal of Systems Architecture*, 54(6):551–561, 2008.

[14] V. Shaiva. Designing adaptive components for a services oriented architecture. *Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on*, pages 390–394, Aug. 2003.

[15] T. Stahl and M. Voelter. *Model-Driven Software Development*. Wiley, 2006.

[16] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison Wesley, 2002.

[17] M. Thonhauser, C. Kreiner, E. Teiniker, and G. Schmolezer. Data model driven enterprise service bus interceptors. In *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, pages 11–18. Euromicro, IEEE Computer Society, September 2008.

[18] M. Thonhauser, G. Schmoelzer, and C. Kreiner. Model-based data processing with transient model extensions. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 299–306, Washington, DC, USA, 2007. IEEE Computer Society.

[19] L. Veiga, R. Rodrigues, and P. Ferreira. Gigi: An ocean of gridlets on a grid-for-the-masses. *Seventh IEEE International Symposium on Cluster Computing and the Grid(CCGrid07)*, 2007.

ECQL: A Query and Action Language for Model-Based Applications

Ulrich Krenn, Michael Thonhauser, Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
michael.thonhauser@TUGraz.at

Abstract

Modern distributed computer systems with mobile and embedded devices as first class citizens are formed from heterogeneous platforms. To support this heterogeneity along with adaptation of the system an approach for interpretation of domain specific models at runtime has been proposed with the concept of Model-Based Software Components (MBSC), separating the domain specific functionality from the current technical platform. This is achieved by the usage of different sets of high-level models. These sets are interpreted by a portable, plugin-extensible runtime environment, utilizing several instances of model-based containers (MCC) for models and their corresponding data.

In this paper the design of a domain specific language is presented, enabling the specification of accessing and manipulating data entities provided by various MCCs used in the runtime architecture of a MBSC. For demonstration purposes the application of the various language elements is presented using a case study of an exemplary distributed pervasive system running in the business domain of logistics.

1. Introduction

According to Tanenbaum a pervasive distributed system is made up of mobile and embedded devices, which are integrated as part of their surroundings [1]. Because of the technical improvements in the last decade performance and storage capacities of these devices are increasing, thus enabling their usage as first class members in pervasive distributed systems. Such devices are not only used for visualization of external data or as producers of raw data, but are enabled to perform some operations directly on the device and to cache processed data locally.

Considering the dynamics of distributed pervasive systems, client functionality needs to be executed on different device platforms, therefore a solution is required to be specified in a technology supporting heterogeneous (hardware) platforms. While some programming language approaches (e.g. Java, .NET) apply virtual machine technology to solve this problem, produced code still has a dependency on the platforms targeted by the implementations of the corresponding VMs. Another solution is proposed by Model Driven Software Development [2] approaches, which are based on the specification of domain specific functionality in a platform independent way applying different models. While mapping to a specific target platform is traditionally done at development time by code generators, requirements for dynamic reconfiguration of pervasive distributed systems foster the interpretation of models at runtime, changing the binding time of the domain specific functionality.

The concept of model-based software components (MBSC) [3] has been proposed to support model interpretation at runtime, using several views on a model as well as the composition with other MBSCs for specification of domain specific functionalities. These views are based on specific models allowing the definition of structural and behavioral concepts of the MBSC. Because at runtime the model and data of the different views are hosted by various model-based data containers (MCCs), referencing data in these containers or performing data manipulation operations demands the specification of these statements in a platform independent (model-based) language.

An approach for such a language is presented in this paper with the EntityContainer Query Language (ECQL), enabling the specification of data and model references as well as arithmetic and logical operations on defined data (sets).

2. Related work

Modern software development approaches are evolving out of Model Driven Software Development (MDSD) [2] techniques to increase the number of supported target platforms. In traditional MDSD approaches like OMG's Model Driven Architecture (MDA), models represent the platform independent specification of developed artifacts, that are mapped to a platform specific representation by code generators at development time. Beside the Unified Modeling Language (UML), other modeling language approaches have been proposed based on the four-level meta-model hierarchy by the OMG, leveraging several layers of models (named M3 to M0) with each layer defining the valid model elements of the next lower layer.

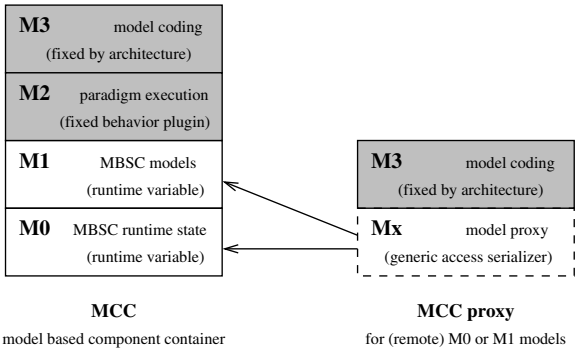


Figure 1. Model layers in MCC and its proxy

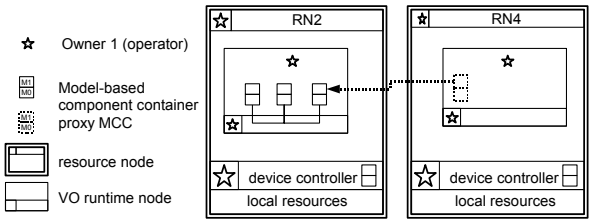


Figure 2. Distributed runtime architecture for MBSCs

UML models are often applied in the system design and development phases focusing on the specification, in contrast to their interpretation at runtime as proposed by Executable UML [4]. Because a lot of views are supported by the UML, its language specification is often criticised for its complexity and generality, impeding the support of domain specific modeling [5]. In contrast, approaches like Domain Driven Design [6] foster the usage of domain specific models – based on a customized and tailored meta-model – throughout the entire lifetime of a system, allowing to use this model as an ubiquitous language between users and developers.

The application of data models at runtime providing the structural specification of the developed software is supported by the concept of the EntityContainer (EC) [7]. Elements defined in the data model are treated as entities being accessible by a dynamic or static interface and having their own identity and lifecycle. The EC has been used as basis for several concepts like Transient Model Extension (TME) [8]. The EC approach has also been extended to become a Model Based Component Container (MCC) supporting state machines model views etc. as well. Fig. 1 depicts the realization of the OMG four-level meta-model hierarchy by a MCC at runtime. Some model layers are treated as being static (i.e. these models are loaded at the instantiation time of an MCC), the lower layers are configured dynamically at runtime. Also, the realization of a proxy using the basic layer of an MCC is demonstrated.

Model-Based Software Components (MBSC) aim for a full model-based definition of the domain specific functionality of a component using several views on the component model [3]. Each MBSC is interpreted by a plug-in driven runtime architecture, which is made up of several MCC instances holding the different views of the domain specific MBSC's. An exemplary distributed system made up by this runtime architecture is shown in Fig. 2.

In this architecture, hardware is represented by resource nodes (RN). They can contain several virtual organization (VO) runtime nodes (VON) defined by the RN owner. Each VON can be used by a (possibly other) owner for executing one or several composed MBSCs. It is given the permission to access specific local resources of the resource node upon its definition by the RN owner. Note that several VONs can be used to realize the dynamics cooperation of different owners as proposed by the concept of Virtual Organizations [9]. Communication between different VONs is realized via a proxy MCC (see Fig. 1). In Fig. 2, the VON within RN4 contains such a proxy to access a model residing at RN2.

2.1. Domain specific languages

Construction of models is performed in a graphical or a textual way, with the latter being supported by a domain specific language toolkit [5]. A domain specific language (DSL) is defined by a meta-model, containing the elements to be used in this language along with a definition of the language syntax. Depending on the domain in focus a DSL can be more general (e.g. UML for describing various parts of a developed software), or very specific (e.g. specifying service composition [10], collaborative systems' development [11]). The design of domain specific languages has been an active area of research in the last years. While Selic discussed the general aspects of DSL design in [12], Spinellis presented different patterns for the construction of DSLs [13].

For the domain of data manipulation several languages have been specified with respect to the data representation technology. While the structured query language (SQL) has become a standard for specifying statements concerning relational databases, several languages (like Hibernate Query Language (HQL) [14] or the Object Query Language(OQL)) have been specified for the definition of queries in object oriented programming languages. SQL is used for specifying data retrieval and manipulation statements, languages in object oriented approaches are often applied for selection [14], leaving other operations to the data container infrastructure used. To specify the selection of entities managed by an EC, the Object Navigation Query Notation (ONQN) has been proposed [15], allowing the selection of trees and graphs by means of specifying a navigation path through the model.

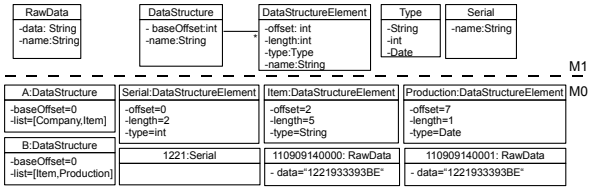


Figure 3. Example contents of MCC "DataContainer"

3. EntityContainer Query Language (ECQL)

ECQL facilitates basic operations on standard data types (strings, numbers, etc.), provides abilities to access and modify data in MCCs and enables the representation of relationships between several data entities like navigating over association connections in a class diagram.

3.1. Language Elements

Several data types are supported by ECQL. Beside the types *Bool, Number, String* and *void* a *Set* type is an important feature of ECQL for enabling various use cases (like applying operations on a number of entities in an MCC simultaneously (i.e. in one ECQL statement). Retrieving a set of entities is done with the help of filters or by applying the *SETREF* statement. Note that all exemplary statements are applied to an MCC containing models and data as depicted in Fig. 3.

3.1.1. Filter functions. While a *VALUE_FILTER* simply returns a set of entities containing values that fulfill the filter criteria, the *ASSOCIATION_FILTER* statement facilitates relationships between several entities, providing the ability to navigate across associations in a class diagram. As visible from the class representing this statement in the meta model (Fig. 4), two sets of entities are required as parameters as well as a string identifying one association's name in the entities of the first set.

```
ASSOCIATION_FILTER(
    SETREF(Entity(DataContainer, M0,
                  Datastructure, *)
    ),
    SETREF(Entity(DataContainer, M0,
                  DataStructureElement, Item)
    ),
    "structure");
```

Listing 1. Using the Association Filter

List. 1 expresses the query *"every DataStructure which contains a mapping for a DataStructureElement with the ID Item"*. The starting set is a list of all entities, which are an instance of the type **DataStructure** and are contained in an MCC named **DataContainer** (Fig. 3). This set is filtered by looking up each associated entity for the association *structure*. Only if the associated entity is of the type **DataStructureElement** and named Item, the currently observed entity is part of the first set in the result.

Considering the possibility of filtering sets of entities before applying the *ASSOCIATION_FILTER* function to them and of even nesting several filters – since all of them evaluate to a set of entities – the support of complex, composed queries becomes apparent.

3.1.2. Reference Functions. Reference functions are required to express data access during runtime interpretation. To achieve a non-ambiguous naming convention, every data is referred by its fully qualified name in the VO resource node. While List. 1 demonstrated the retrieval of an entity set, List. 2 instructs the runtime environment to access a number value named *valueEntry.baseOffset* in entity *A* which is of type *DataStructure*. The entity is looked up at the data layer (M0) of MCC *DataContainer*.

```
NUMBERREF(Entity(DataContainer,M0,
                 DataStructure,A),
          Attribute(valueEntry,baseOffset));
```

Listing 2. Reference Functions in ECQL

Type safety checks for references can only be executed at runtime (interpretation time) since data referred to is not necessarily available at insertion time of the ECQL statement.

3.1.3. MCC manipulation. Quite a similar syntax is used to express MCC data manipulation operations (for various layers) in ECQL. The statement shown in List. 3 instructs the runtime environment to insert a new number value entry – equivalent to attributes in the UML – named *length* into entity *Item* of type *DataStructureElement*.

```
INSERT (Entity (DataContainer, M0,
                DataStructureElement, Item),
        Attribute (valueEntry,length), 2);
```

Listing 3. Data Manipulation with ECQL

Again, the entity resides in the layer *M0* of the MCC named *DataContainer*.

3.2. Statement signatures

Each ECQL statement has a well-defined signature which facilitates type safety checks for any ECQL statement inserted into an MCC.

```
Bool SMALLER_THAN(Number first, Number second);
Bool BOOLREF(Entity entity, Attribute attribute);
Set INTERSECTION(Set first, Set second);
```

Listing 4. ECQL Signatures

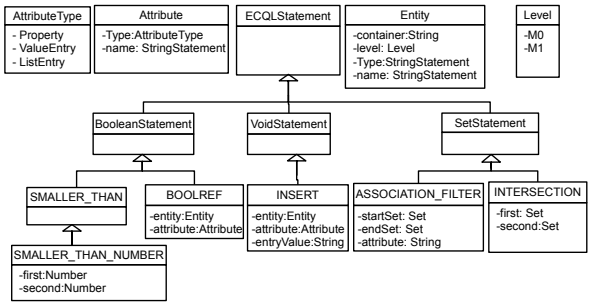


Figure 4. Partial ECQL metamodel

List. 4 gives an example of some signatures derived from the meta-model depicted in Fig. 4. Meta-model elements in the first stage of the inheritance tree are defining the return type while inherited elements specify the name of the statement as well as the number of parameters.

3.3. Platform extension with ECQL

Fig. 5 gives an overview about the relevant models and the operations applied on the MCC in the concept of ECQL. At creation of an MCC the corresponding M2 model is loaded and is treated as static during runtime (c.f. Fig. 1). For effective use of ECQL statements inside an MCC (e.g. as state chart actions, for databinding purposes) the M2 model of the MCC has to be extended with the ECQL meta model. Such an extended meta model enables a full validation of loaded M1 models with embedded ECQL statements also. Extension of the meta model is done with a Transient Model Extension (TME) [8], allowing the insertion of new model elements and the connection of existing model elements based on the rules provided in the next upper layer (in this case the M3 layer). A second extension in the M2 layer is performed during the instantiation of a MCC, allowing the support of assigned resources to a VON (e.g. reading RFID tags, using hardware's integrated speaker) with additional ECQL statements. Using this approach enables model based checking of resource constraints at runtime by the MCC, leading to an abortion of the loading procedure in case the MCC respectively the VON have not been assigned the required resources.

```
INSERT (Entity (DataContainer, M0,
                RawData, getTimestamp()),
        Attribute(valueEntry, data),
        getRfidCode());
```

Listing 5. Using Platform Defined ECQL Statements

List. 5 demonstrates this feature by presenting a statement to add a new entity of type *RawData* to the MCC **DataContainer**. This MCC is contained in a VON running on an RFID reader acting as the RN. The owner of this RN

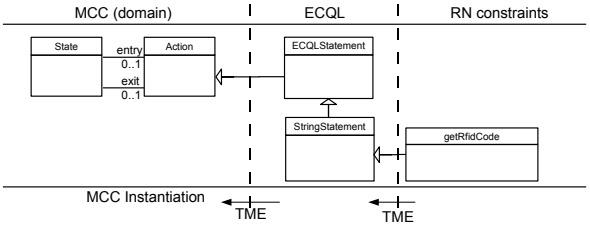


Figure 5. Extension of ECQL at runtime

has granted the right of using the data of the currently read tag to the MBSC being executed in the VON. This right is represented by the ECQL statement **getRfidCode()**, which is added to the ECQL meta model of the VON and thus to the M2 layer of all MCCs in this VON. In the same way an ECQL statement for requesting a timestamp from the RN is added to the container, and is used for determining the name of the new entity instance in the exemplary ECQL statement. If the resource of reading RFID tags is not made available to the VON, the corresponding ECQL statement is missing in the MCC DataContainer and thus the validation of the ECQL statement in List. 5 fails.

4. Example

The presented approach has been evaluated in a scenario demonstrating shared usage of an RFID reader in a distributed system. This RFID reader is located in the incoming goods division of a logistic provider's distribution center. Various computer parts labeled with RFID tags are arriving from different suppliers and have to be delivered to a retailer of computer hardware.

The functionality for detecting specific goods, which have been marked by the retailer (e.g. to perform a quality check for a distinct supplier upon receiving the computer parts in the distribution center), has been implemented using the MBSC approach. Note that Fig. 2 depicts the deployment situation of this MBSC, running on a VON on the RFID-reader hardware (RN4), while another MBSC is running on a centralized server (RN2). The data model and a snapshot of the data contained in the MCC named DataContainer in the distinct VON of RN4 is depicted in Fig. 3.

Fig. 6 shows a filter behavior of RFID tags read by the RN4. The state machine starts in *wait for RFID*. Every time an *RFID available*-event is fired by the hardware, the state machine enters the *RFID available*-state and proceeds to the processing state where the tag data gets split into several fields according to the data model. In case the tag's serial is found in the list of serial numbers transmitted earlier, the state machine enters the state at the bottom of Fig. 6 where the tag data is transmitted to the server through a proxy MCC as already shown in Fig. 2.

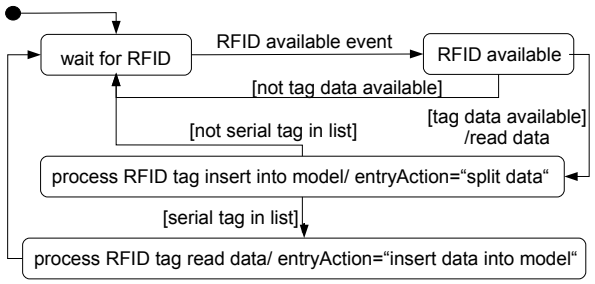


Figure 6. State Chart for Example Scenario

In the state chart for the prototype four ECQL statements are specified for the conditions, which are marked with squared brackets at the transitions. One ECQL statement is specified as the action *read data* of a transition, and two ECQL statements are used for the entry actions of the corresponding states. Both *split data* and *read data* statement make use of RN specific ECQL statements, which are realized by a RFID reader specific plug-in of the VON.

The meta model presented in Fig. 4 has been evaluated with an editor for the presented DSL implemented using the Eclipse modeling project [5]. The language definition and scenario has been implemented in a prototype on top of the .NET Microframework. JSON has been used for the persistent storage of models and data handled by the MCCs in the prototype, enabling a lightweight possibility for interoperability with other programming language technologies.

5. Conclusion

In this paper a domain specific language for querying and manipulating the data and models of a model-based component container (MCC) has been presented. Several MCCs are applied for interpretation of a Model-Based Software Component (MBSC) at runtime, which captures the domain specific functionality in a set of high-level models. During initialization of a MCC the static metamodel is extended with the ECQL meta-model and additional ECQL statements provided by the VON containing the instantiated MCC.

The ability for filtering and retrieving entity references from an MCC via ECQL statements has been discussed as well as the application of several MCC manipulation statements. The given language has been evaluated in a prototype realizing a scenario in the business domain of logistics featuring the execution of MBSCs directly on an RFID reader hardware.

Acknowledgment

We would like to thank Manuel Menghin and Nicolas Pavlidis for their support in the development of the tooling. This work was funded by RF-iT Solutions GmbH and FFG.

References

[1] A. S. Tanenbaum and M. van Steen, *Distributed Systems. Principles and Paradigms*, 2nd ed. Prentice Hall International, October 2006.

[2] M. Voelter and T. Stahl, *Model-Driven Software Development*, 1st ed. Wiley & Sons, May 2006.

[3] M. Thonhauser, C. Kreiner, and M. Schmid, "Interpreting Model-Based Components for Information Systems," in *Engineering of Computer Based Systems, 16th Annual IEEE International Conference and Workshop*, 2009, pp. 254–261.

[4] D. Milicev, *Model-Driven Development with Executable UML*, 1st ed. Wrox, July 2009.

[5] R. C. Gronback, *Eclipse modeling project : a domain-specific language toolkit*, 1st ed. Addison-Wesley, April 2009.

[6] E. J. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1st ed. Addison-Wesley Professional, September 2003.

[7] G. Schmoelzer, S. Mitterdorfer, C. Kreiner, J. Faschingbauer, Z. Kovács, E. Teiniker, and R. Weiss, "The Entity Container - An Object-Oriented and Model-Driven Persistency Cache," in *HICSS*. IEEE Computer Society, 2005.

[8] M. Thonhauser, G. Schmoelzer, and C. Kreiner, "Model-based Data Processing with Transient Model Extensions," in *Proceedings of the 14th Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*. Tucson, AZ, USA: IEEE, March 2007, pp. 299–306.

[9] M. Thonhauser, C. Kreiner, and A. Leitner, "A Model-Based Architecture supporting Virtual Organizations in Pervasive Systems," in *15th IEEE International Conference on Engineering of Complex Computer Systems*, 2010, to be published.

[10] S. Haschemi and A. Wider, "An Extensible Language for Service Dependency Management," in *Software Engineering and Advanced Applications, 2009 (SEAA'09). 35th EUROMICRO Conference on*, EUROMICRO. IEEE CS, 2009.

[11] L. M. Bibbo, D. Garcia, and C. Pons, "A Domain Specific Language for the Development of Collaborative Systems," in *Int. Conf. of the Chilean Computer Science Society*, November 2008, pp. 3–12.

[12] B. Selic, "A Systematic Approach to Domain-Specific Language Design Using UML," in *10th IEEE Int. Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE Computer Society, May 2007, pp. 2–9.

[13] D. Spinellis, "Notable design patterns for domain-specific languages," *Journal of Systems and Software*, vol. 56, no. 1, pp. 91–99, February 2001.

[14] C. Bauer and G. King, *Hibernate in Action (In Action series)*, illustrated edition ed. Manning Publications, August 2004.

[15] S. Mitterdorfer, E. Teiniker, C. Kreiner, Z. Kovács, and R. Weiss, "A New Design Of An Object Navigation Query Notation," in *International Conference on Software Engineering and Applications (SEA)*. Calgary, AB, Canada: ACTA Press, 2002, pp. 156–161.

MODEL BASED DATA ACCESS IN MOBILE GRID APPLICATIONS

Michael Thonhauser[1,2], Christian Kreiner[1,2], Stefan Kremser[1]
[1]Institute for Technical Informatics
Graz University of Technology, Austria
Inffeldgasse 16, A-8010 Graz
michael.thonhauser@TUGraz.at
[2]Salomon Automation GmbH
Friesach bei Graz
A-8114 Friesach, Austria

ABSTRACT
Development of data intensive systems has changed from applying relational database concepts to model-based development technologies. A central aspect of these approaches is the design of a data model, which can be used for creating the persistent datastructures.

While approaches following the Model Driven Architecture use the model for code generation at development time, other approaches interpret the model at runtime, which enables dynamic access to modeled data.

On the other hand arises the need for managing data access in mobile applications, regarding concerns of privacy and data integrity. By relying on a mobile grid infrastructure, the central concept of a virtual organisation can be extended to enable the exchange of datamodels between the members of this VO, containing the externally accessible datastructures of the publishing member.

This paper presents an extension of a model-based approach for data access in a data intensive system, which enables mobile devices to dynamically use data of such a system.

KEY WORDS
Model-Based Development, Mobile Grid, Data-intensive systems, logistic systems

1 Introduction

Data intensive applications are software systems that focus on data processing, data visualization and data storage (such as enterprise resource planning systems, banking applications or logistic systems) often relying on large and complex data structures. Such applications are already developed using network technologies to connect local fixed workstations to the database server.

In the last years there has been a lot of research on data intensive distributed systems consisting of mobile devices, which has led to different hardware platforms like Smartphones and Personal Digital Assistants (PDA) and different wireless communication technologies such as Wireless LAN, Bluetooth, Radio Frequency Identification (RFID) and ZigBee. This development is driven by the vision of ubiquitious computing [1].

Because every mobile device gets more and more self aware of its context and data, applications have to deal with an increasing number of distributed data [2]. While in the past data intensive applications have worked with big databases, modern software is written to deal with data collected by different sensors and being stored and analyzed in real time.

Writing software for distributed systems requires a lot of consideration on data security and data integrity [3]. Many mobile application development approaches are missing an interface for the user, to control the data being broadcasted by the application. Often this data is held in a data structure being defined in the currently used programming language. On the other hand there are trends like Model Driven Development [4], which make use of models describing the data structures and methods used by an application.

To integrate these new requirements in mobile applications accessing data intensive systems, this paper is going to introduce a model based software component extending the concept of an object oriented model driven data cache [5]. Developed for easing the communication of stationary clients with a database, this concept is going to be extended to use the information and technologies available in a mobile and distributed environment.

The rest of this paper is organized as follows: Section 2 introduces the concepts of MBD and grid computing. The extension of the object oriented model-driven cache is presented in section 3. Section 4 contains an example outlining aspects of applying the extended component.

2 Related work

2.1 Model Driven Development

Model Driven Development (MDD) [6] is an approach to design a software system by describing it in a Platform Independent Model (PIM). A PIM defines associations between the data and the behavior of the software and is used as input for generators producing a platform specific model (PSM). To support MDD the Object Management Group (OMG) has released the Model-Driven Architecture (MDA) containing standards, that enable the specification and transformation of models.

2.1.1 Data modeling

Models of software design are often specified using Unified Modeling Language (UML) , another standard of

597-080

the OMG. UML models are based on a metamodel and are situated in the user model layer of the four-level metamodel hierarchy [7]. UML describes several diagrams, which can be used to model different aspects of the software.

The class diagram is used to model the structure of classes, such as attribute and methods, as well as the association between the different classes of a program. The metamodel for a class diagram can be extended for data modeling purposes, focusing on class attributes and associations between the different classes.

2.1.2 Agile Model Driven Development

Agile Data Modeling [8] relies on iterative construction of data models for the modeled software, where each data model fits to the requirements needed in the actual iteration. It fits best to applications, that rely on relational databases for persistent data storage. Agile Model Driven Development (AMDD) also uses an iterative approach, instead of extensive models being generated in the normal MDD process.

2.2 Grid Computing

One definition of Grid Computing is given by Ian Foster and Carl Kesselman, stating that to solve a scientific problem together, the distributed resources of scientists within different administrative domains can be dynamically and coordinately connected by using fast networks to build a virtual computing center/organization (VO) [9]. This use case is typically realized by Computing Grids and Data Grids [10].

While these types of grids are focussed on connecting large powerful computers, resource grids additionaly provide access to dedicated rare resources (like a hydron collider or a space telescope)[11]. Considering the fact that mobile devices have become more powerful during the last years, they are also targeted by different grid approaches. While wireless grids [12] are focused on building grids up of mobile devices such as laptops, pervasive grids [13] try to integrate mobile devices and sensor networks in the fixed grid infrastructure. Another term used in the literature are mobile grids [14, 15], which can be seen as an evolutionary step between wireless and pervasive grids.

2.3 MDD and Grid Computing

Smith et al. present an approach for MDD of service oriented grid applications in [16]. Their approach makes use of an UML profile extending the UML *Class*, *Attribute* and *Operation* metaclasses with *GridClass*, *GridAttribute* and *GridMethod* stereotypes. This profile is applied to UML models to define grid service interfaces, which are generated in a sublayer of the PSM model. Code is generated for the Java programming languages, and stereotypes of the UML profile are mapped to Java annotations.

Compared to our approach, this approach provides a more static usage of the data model, because it is used for code generation purposes at development time, while our approach interprets the extended data model at runtime.

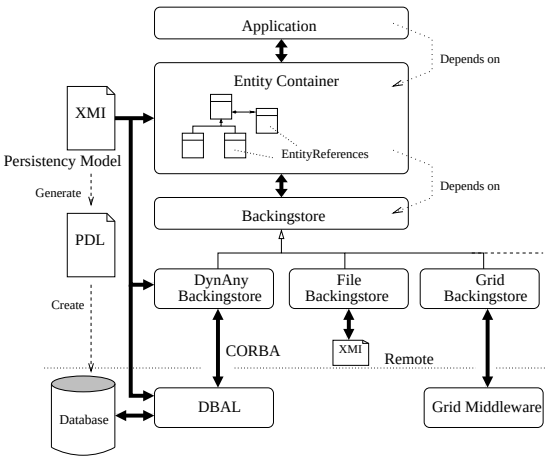


Figure 1. Entity container architecture

3 Framework for mobile applications

3.1 Entity Container

Provided that the structure of the persistent data has been modeled, an **Entity Container** (EC) [5] can be used as a model-based object oriented data cache. The architecture of the EC is shown in Fig. 1. The EC provides distinguishable objects called entities, identified by a unique value. It operates on two levels of the four-level metamodel hierarchy of OMG implementing the *instance of* relation between these two levels.

Usually the UML data model is stored in a file using the XML Metadata Interchange (XMI) format. This file contains the UML model of the persistent data, which is itself based on the UML metamodel, which is extended with a profile. The data model is used by the EC and the associated backingstore.

There exist different implementations of the backingstore interface, such as an object-relational bridge (DBAL) for using relational databases, a XML filereader and writer, and an inmemory backingstore. Data entities in the EC are accessed using a dynamic interface. In this MDD approach the database is created from the persistency model, which is also used to configure the EC and its associated backingstore.

The EC has been designed to support the client-server communication paradigm as well as the offline use case of a data intensive client application.

3.2 Grid BackingStore

To integrate a new communication paradigm, such as peer-to-peer communication between two EC using applications, another implementation of the backingstore interface is needed. Fig. 2 illustrates the components used for building such a backingstore.

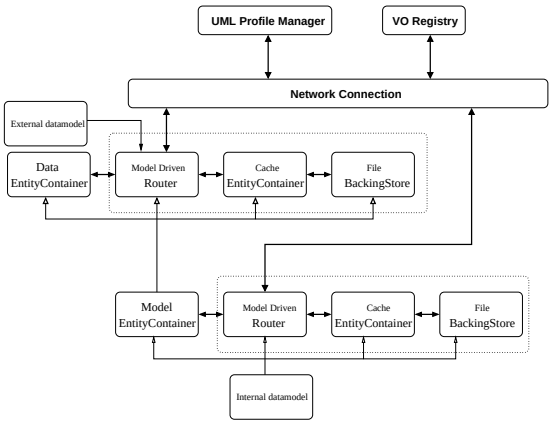


Figure 2. Grid BackingStore architecture

Model Driven Router: A model driven router (MDR) provides two interfaces. The first interface is used by the EC, which is connected to the Grid BackingStore(BS). This interface is used for synchronizing the local data cached in the EC with the internal EC of the Grid BS. The second interface is used for communication between the Grid BS and the grid middleware.

While querying the network has been mentioned before, the other use case consists of processing queries from the network. In this case the MDR looks up, if a requested class is available for outside processing by looking at the external data model. This data model defines for each class, if it is used for *input*, *output* or *input/output* purposes. Additionally it can define constraints, which need to be fullfilled by the requesting client (e.g. being connected via a secure connection or using a broadband network). These constraints are defined in the GridProfile, which is available from the VO management server.

Cache EC: The internal EC is used for enabling persistent storage of locally generated data and it is also used for resolving queries from the connected devices in the network.

File Backingstore: This backingstore contains the persistent local data of the application.

Note that the architecture depicted in Fig. 2 illustrates the data layer (M0) and model layer (M1) of the four level metamodel hierarchy defined by the OMG [7].

Therefore it is possible to build up a data model dynamically by querying the Model EC only for the model elements currently needed. As mentioned before the datamodel contains *EXTERN* references, which define, that the structure of this class is retrieved at runtime from the connected Grid members using the second level Grid BS. Because external references are only resolved if needed, the initial data model can be very small, because the structure of many classes is defined by external models.

3.3 Framework Overview

Fig. 3 illustrates the essential components of the proposed framework for model based development of distributed applications. The framework is used by a fixed device (e.g. a UNIX server machine) and by a mobile device, which is able of capturing sensor data (e.g. temperature or lighting conditions). An application is executed on each device, which makes use of an EC. The EC used by each application is configured with a GRID backingstore, which has the following tasks:

1. Enable access to a virtual organisation (VO) via a grid middleware.
2. Provide the classes available in the external data model for other members querying for external datastructures in the Grid as well as the corresponding data.

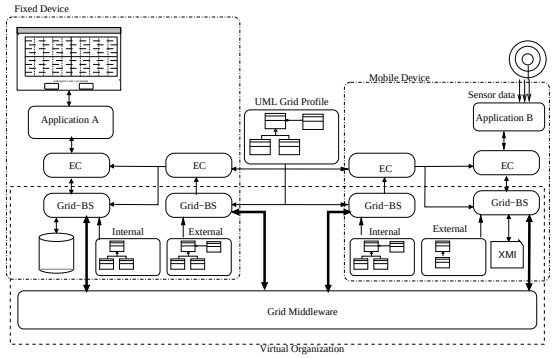


Figure 3. Applied Grid Backingstore

4 Example

For demonstrating our approach an exemplary virtual organization (VO) is created, which contains mobile devices being part of a supply chain. The VO is used to manage incoming deliveries of suppliers for a warehouse. According to Fig. 4 following types of members are differentiated:

1. A mobile client (a PDA or mobile phone) is used by the supplier to create an order containing his delivered transport units (**TU**).

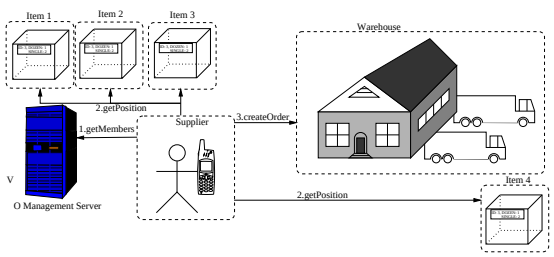


Figure 4. Members of the VO

2. This order is stored and processed in a warehouse management system (WMS).

3. Each TU is equipped with an intelligent interface (such as an RFID label or a mote), and is connected to the VO.

The WMS is constructed using the data model depicted in Fig. 5. Note that this data model defines two systems. In both systems the content of a **TU** is modelled by a **TUContent** class. Additionally to System 1 the **TUContent** class of System 2 has the attributes *Batch* and *BestBefore*.

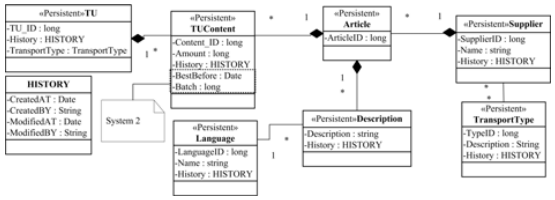


Figure 5. internal data model of WMS

Based on the model in Fig. 5 the WMS developer or the warehouse operator creates an *external data model*, which is shown in Fig. 6. Classes contained in the model have applied a stereotype, which defines whether the class is available for *input*, *output* or *input/output* purposes. Additional elements in the model can be configured with different constraints, regulating the access to data being instances of that element.

The mobile application of the supplier is developed using the data model presented in Fig. 7. This data model is created by the developer of the mobile application. Note that classes in this model contain the **EXTERN** stereotype meaning that the structure of the corresponding class (e.g. its attributes) is loaded at runtime from the currently connected WMS.

Once the application has been started on the mobile device by the supplier, it looks up the other members of the VO, such as warehouses and other mobile devices. The

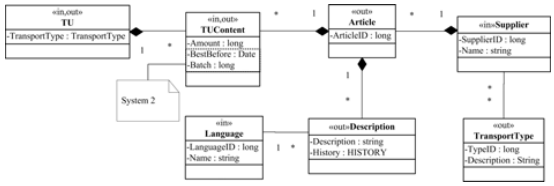


Figure 6. external data model of the WMS

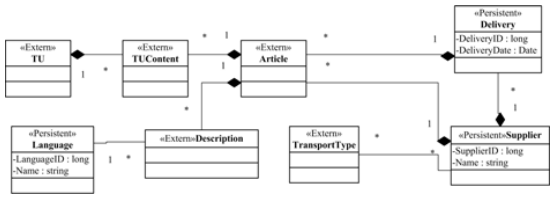


Figure 7. internal data model of mobile application

data for the displayed dialogs is retrieved using an EC configured with a Grid BS.

If the supplier decides to create a new delivery, a list of all warehouses connected to the grid is presented to the supplier. Having selected a specific warehouse a new dialog is displayed allowing the supplier to add the articles he is going to deliver to the selected warehouse. While loading the dialog, the Grid BS has resolved the *EXTERN* marked classes with the corresponding classes defined in the external data model of the selected warehouse.

Additionally to manual input of the delivered goods, the supplier can also access goods containing an intelligent RFID chip, which is also connected to the grid.

Having finished the delivery creation procedure, the supplier synchronizes the delivery specific data on arrival at the warehouse. Because he got the WMS specific attributes of the TUContent class, the data is easily imported in the WMS.

5 Implementation

Our current work considers the evaluation of the presented architecture in Sect. 3 using the example presented in Sect. 4. The first step has been the refactoring of the current EC implementation, because it has been developed using the C++ programming language. This implementation is made accessible to Java programs using a Java Native Interface (JNI). Since JNI is not supported by the current J2ME specification, we refactored our toolchain, to support direct generation of J2ME compatible code for the metamodel layer needed by the EC.

Alongside we also started to evaluate Grid middleware frameworks to find out their adaptability to mobile devices. We started with a survey of existing middlewares.

System 1 System 2

Figure 8. Dialogs for adding article to delivery

Some, like GridGain [17], GridKit [18], the Grid Application Toolkit(GAT) [19], and Globus Toolkit [20] looked promising at a first glance. The first two rely on $Java1.5$ and above, thus making it impossible to use them on a J2ME platform as they are. The GAT project was completed in April 2005, so further development is not to be expected.

The next approach would have been the use of Globus Toolkit and its webservice-infrastructure. Two considerations have to be made though: Implementing a WSRF- or webservice-client on J2ME implies the use of the J2ME Web Services Specification(JSR 172) or some third-party library. However, we want an application that is highly portable on most available Java enabled mobile phones and further we would like to avoid the communication overhead of web services.

In the end we decided to use the Java Agent Development Framework(JADE) for our prototype application, since there are implementations for nearly every Java environment(J2EE, J2SE, J2ME) providing the same API. One advantage we were looking for, are light-weight clients and JADE seems to provide us with this requirement [21].

5.1 Scenario evaluation

For evaluation of the examplary scenario from Sect. 4 a prototype is being developed using J2ME, because this platform is available in nearly every mobile phone. Fig. 8 depicts the dialogs for adding an article in the delivery planning application of a supplier, which is based on the datamodel presented in Fig. 5 and Fig. 7 respectivly. The right userinterface is based on the datamodel of System 2 in Fig. 5, the left userinterface shows the dialog for System 1, thus missing the fields for the batch and bestbefore date.

While these userinterfaces are now implemented using a static approach, we are considering a dynamic implementation using the model information provided by the Grid BS. Another plan is to provide support for the .NET platform, which is widely used by PDAs.

6 Conclusion

We have presented an architecture for model based data access of mobile devices. Our approach is based on the conceptual extension of a model-driven object oriented data cache, which is used for accessing a database by applying dynamic interfaces. This cache is coupled with an implementation of the BackingStore interface, which is used for persistently storing the data of the cache. Because the basic datamodel is interpreted during runtime by the EC and its associated backingstore, the same infrastructure is used for processing an external data model. This data model is constructed from the basic data model, and contains the interface specifications as well as quality of service constraints for other devices in the mobile grid, which want to get access to the data contained in the GridBackingStore. While our approach allows the software developer to access data in a transparent way regarding the persistent storage and grid technology, it gives the user control over the data provided to other grid members via the external datamodel. While our ongoing work consists of finalizing the implementational prototype, we are also considering the development of model-driven user interfaces, to ease the control of provided data structures by the user.

References

[1] M. Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(3):3–11, 1999.

[2] M. Denny, M.J. Franklin, P. Castro, and A. Purakayastha. Mobiscope: A scalable spatial discovery service for mobile network resources. In *MDM '03: Proceedings of the 4th International Conference on Mobile Data Management*, pages 307–324, London, UK, 2003. Springer-Verlag.

[3] T. Abdelzaher, Y. Anokwa, P. Boda, J. Burke, D. Estrin, L. Guibas, A. Kansal, S. Madden, and J. Reich. Mobiscopes for human spaces. *Pervasive Computing, IEEE*, 6(2):20–29, 2007.

[4] B. Selic. Model-Driven Development: Its Essence and Opportunities. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006.*, page 7pp. IEEE, April 2006.

[5] Gernot Schmoelzer, Stefan Mitterdorfer, Christian Kreiner, Joerg Faschingbauer, Zsolt Kovács, Egon Teiniker, and Reinhold Weiss. The Entity Container

— an Object–Oriented and Model–Driven Persistency Cache. In *HICSS'05, Big Island, Hawai'i, USA, Jan. 3-6*, page 277b. IEEE, 2005.

[6] Stephan J. Mellor, Anthony N. Clark, and Takao Futagami. Model–driven development — guest editor's introduction. *Software, IEEE*, 20(5):14–18, 2003.

[7] OMG. UML Infrastructure, Version 2.0 . Technical Report 2002-09-01, Object Management Group, 2002.

[8] Scott W. Ambler. *Agile Database Techniques*. Wiley Publishing, Inc., 2003.

[9] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[10] R. Moore, A.S. Jagatheesan, A. Rajasekar, M. Wan, and W. Schroeder. Data Grid Management Systems. In *Proceedings of the 21stI EEE/NASA Conference on Mass Storage Systems and Technologies (MSST)*, College Park, Maryland, USA, April 2004. EEE/NASA.

[11] I. Foster and C. Kesselman. *"The Grid. Blueprint for a New Computing Infrastructure"*. Morgan Kaufmann, 2 edition, August 2004.

[12] Lee.W. McKnight, J. Howison, and S. Bradner. Guest editors' introduction: Wireless grids–distributed resource sharing by mobile, nomadic, and fixed devices. *IEEE Internet Computing*, 8(4):24–31, 2004.

[13] S.H. Srinivasan. Pervasive wireless grid architecture. In *Wireless On-demand Network Systems and Services, 2005. WONS 2005. Second Annual Conference on*, pages 83–88, 19-21 Jan. 2005.

[14] M. Waldburger and B. Stiller. Toward the mobile grid: Service provisioning in a mobile dynamic virtual organization. Technical report, University of Zurci, Department of Informatics, 2005.

[15] K. Ohta, T. Yoshikawa, T. Nakagawa, and H. Inamura. Design and implementation of mobile grid middleware for handsets. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, volume 2, pages 679–683Vol.2, 20-22 July 2005.

[16] M. Smith, T. Friese, and B. Freisleben. *Grid Computing*, chapter Model Driven Development of Service-Oriented Grid Applications. vieweg, March 2006.

[17] Gridgain. http://www.gridgain.com/. last visited: 11/01/2007.

[18] P. Grace, G. Coulson, G. Blair, L. Mathy, W. Yeung, W. Cai, D. Duce, and C. Cooper. Gridkit: Pluggable overlay networks for grid computing. In *Proceedings of Distributed Objects and Applications*, October 2004.

[19] Gridlab: Grid application toolkit(gat). http://www.gridlab.org/. last visited: 11/29/2007.

[20] The globus toolkit. http://globus.org/. last visited: 11/29/2007.

[21] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE-A White Paper. *Telecom Italia EXP magazine*, 3(3), 2003.

Model-based Data Processing with Transient Model Extensions

Michael Thonhauser[1,2], Gernot Schmoelzer[1,2], Christian Kreiner[1,2]

[1]Institute for Technical Informatics
Graz University of Technology, Austria
Inffeldgasse 16, A-8010 Graz
michael.thonhauser@TUGraz.at

[2]Salomon Automation GmbH
Friesach bei Graz
A-8114 Friesach, Austria

Abstract

Software is often constructed using a layered approach to encapsulate the functionality in different layers. Individual requirements of each layer demand layer specific data structures. These data structures typically provide redundant information with respect to the data source.

Providing a Model Driven Software Development approach for creating these data structures leads to overlapping data models, each containing data structures defined by the data source. Because putting all various requirements of the software layers in a single data model can lead to difficulties, each software layer should only extend the basic data source model with its specifically needed model elements.

This paper presents a mechanism for transient extension of a data model. Using this mechanism, a basic data model can be used by every layer, being extended by additional attributes and classes for satisfying layer specific requirements.

Keywords: Model-driven development; Data modeling

1. Introduction

Data-intensive systems are characterized by manipulating and displaying a large amount of structured data. This data is saved in some persistent storage such as a database or a file. Such data-intensive systems are often designed in three layers [6]. including *presentation layer*, *domain* (or business function) *layer* and *data source layer*.

Information contained in the data source is accessed through the data source layer. This layer transfers data contained in persistent data structures such as database tables, into dynamic data structures, such as arrays or object composites. The transfer functions can be coded manually or can be generated using a data model of the persistent data[1].

Because the problems to be solved by data-intensive software are quite complex, manual programming of transfer functions can quickly lead to logical errors. Bran Selic describes this problem in [14], distinguishing between essential and accidental complexity of software. Essential complexity is inseparable from the problem, but accidental complexity is a direct consequence of the chosen problem solution. For reducing accidental complexity the problem should be looked at an abstract viewpoint, ignoring program language specific details.

This abstraction can be provided by a plattform independent model (PIM), which is created during a Model Driven Development (MDD) process. While MDD tries to capture all aspects of a program using different variants of modeling, like activity diagrams or class diagrams, agile model driven development (AMDD) has its focus on the aspect of a model currently needed. In data-intensive application development one of the first modeled aspects of each layer is its data structure.

2. Motivation

2.1. Layered software

In the previous section the idea of software layers has been introduced. Each software layer internally uses data structures, holding the needed data of this layer. Fig. 1 shows this data structures and depicts the dependencies between them. Every data structure can be seen as a layer specific view on the information generally available to the program. Data structure $S1$ holds the transformed persistent data, which is compliant to the structure of the data source.

$S2$ repesents the data structure used by the domain layer and $S3$ is contained in the presentation layer.

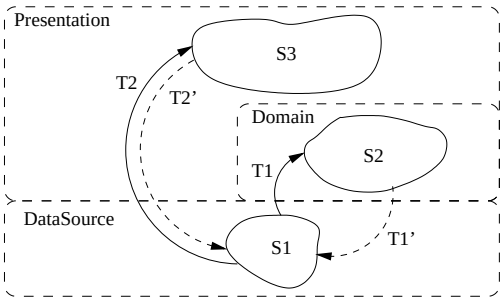


Figure 1. Data structures in software layers

Fig. 2 illustrates an example use case from the business domain of logistics. Boxes containing small items are handled by a warehouse management system (WMS). The WMS is used for checking the amount of items contained in the box, before the box leaves the warehouse. The user interface consists of a table, which additionally displays the number of dozens and single items in a box, to ease the quality check of the boxes. It also makes use of a business function calculating the weight of each box for issuing an alert, if the weight of all boxes reaches the weight limit for transportation.

This example illustrates different requirements for data structures in various application layers. Requirements for $S1$ are the reduction of duplicate information, thus providing support for *data normalization* [1]. This reduction is needed for optimal storage usage, efficient data transfer and to avoid inconistency of the stored data. Also the data structure should be easily mappable to the persistent data source.

$S2$ is driven by requirements for *simplifying complexity* of the application logic, adding *transaction handling* and *data consistency checks*. This aims at avoidance of inconsistent data and is done through *adding data redundancy*.

$S3$ contains the data displayed in the presentation layer. Its content meets user requirements and *usability aspects*, which often requires displaying additional information.

As illustraded in Fig. 1 there exist several dependencies of the data structures in the different layers. Because $S1$ holds all the information available to the application, additional data in $S2$ and $S3$ is normally related to data contained in $S1$. This requirement can be fulfilled by extending $S1$ with a transformation $T1$ or $T2$.

The way this transformation is performed depends on the abstraction level used for application development.

- The WMS described in Fig. 2 can be implemented using a relational database and a structured programming language like C. This programming language can work with a database manipulation language (DML) like SQL to define the data structures $S1$, $S2$ and $S3$. In this case the transformation $T1$ and $T2$ of the data structures is done by extending the DML statement.

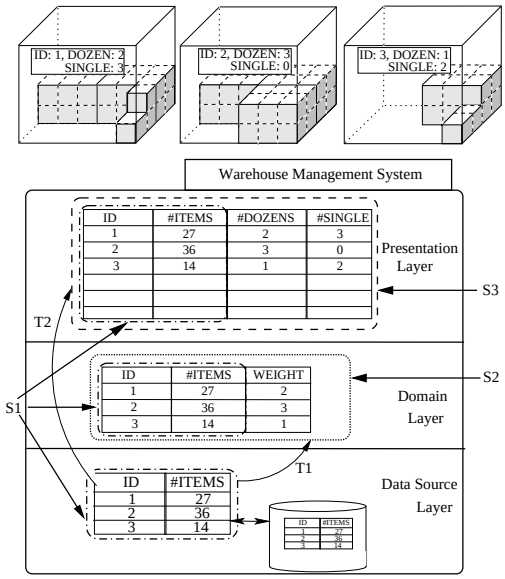


Figure 2. Warehouse management example

- A higher level of abstractions is provided by object oriented languages combined with object-relational methods [6]. This solution enables type safety for $S1$, $S2$ and $S3$ consisting of objects. The transformation is done with additional source code for the implemenation of $S2$ and $S3$.

- A further abstraction level is reached by using a MDD approach. This leads to a correspondend data model $DM1$ for $S1$, $DM2$ for $S2$ and $DM3$ for $S3$. Following the dependencies of the different data structures, the dependencies of the data models can be seen as $DM1 \subseteq DM2$ and $DM1 \subseteq DM3$. In this case transformation $T1$ or $T2$ is done by extending $DM1$. This extension adds information about additional data needed in the presentation layer leading to $DM3$ and to $DM2$ respectivly in the domain layer.

  We call this transformation mechanism Transient Model Extension (TME). Since $DM1$ is used as the basic model for this mechanism, $DM2$ and $DM3$ does not need to be stored in the corresponding layers. Only

the transformation rules needed for construction of the appropriate data model need to be saved instead.

2.2 Model Driven Development

Model Driven Development (MDD) [9] is an approach to implement a software system by describing it in a Platform Independend Model (PIM). A PIM defines associations between data and behavior of the software and it is used as input for generators producing a platform specific model (PSM). To support MDD the Object Management Group (OMG) has released the Model-Driven Architecture (MDA) containing standards allowing specification and transformation of models. Another dominant approach to MDD are Software Factories, which are proposed by Microsoft and can be seen as a new software development paradigm. Differences of these two approaches are discussed in [4].

Data modeling

Models of software design are often specified using Unified Modeling Language (UML) [11], another standard of the OMG. UML models are based on a metamodel and are situated in the user model layer of the four-level metamodel hierarchy [10]. UML describes several diagrams, which can be used to model different aspects of a software. Structural and behavioral diagrams are differentiated. One example of a structural diagram is the class diagram. It is used to model the structure of classes, such as attribute and methods, as well as the association between the different classes in the model.

For data modeling purposes the metamodel of a class diagram can be extended focusing only on class attributes and associations [1].

Agile Data Modeling relies on iterative construction of data models, where each data model satisfies the requirements needed in the current iteration. It is best suitable for applications, that rely on relational databases for persistent data storage. Agile Model Driven Development (AMDD) also uses an iterative approach, instead of extensive models being generated in the normal MDD process.

3. Transient Model Extension

The structure of a data model can be based on the four-level metamodel hierarchy of the OMG, whereas level M3 and level M2 are the same for each data model. Model level M1 contains the domain specific data model, holding the domain specific classes and associations. This elements are used for construction of objects which are contained in model level M0. Generally speaking every model can be seen as an instance of the model in the lower layer. Considering this fact, it is obvious that changing the model in level M1 results in additional data contained in level M0.

Fig. 3 shows the four-level metamodel hierarchy with respect to the data structures defined in Fig. 1. According to this figure $DM1$ is part of every data model used by the different software layers. Data model extensions (DME) correspond to the defined mappings $T1$ and $T2$ in Fig. 1.

$DM2$ is the data model used by the domain layer. It is defined as a combination of $DM1$ and $DME2$. In the same way the data model of the presentation layer ($DM3$) results from combining $DM1$ with $DME3$.

Applying the AMDD approach to a layered software architecture results in the definition of the data model describing the data structures used by the data source layer. In the next steps the corresponding data models for the domain and the presentation layer are defined. According to Fig. 3 a separated definition of each data model used by the different software layers is not feasible. Identical information needed by different layers is duplicated in the models, leading to maintenance difficulties of the shared information. Also each layer needs to store its own data model, which can lead to increasing storage requirements by the application. To overcome these problems a data model for all layers could be used, that is adapted for the requirements of the different layers.

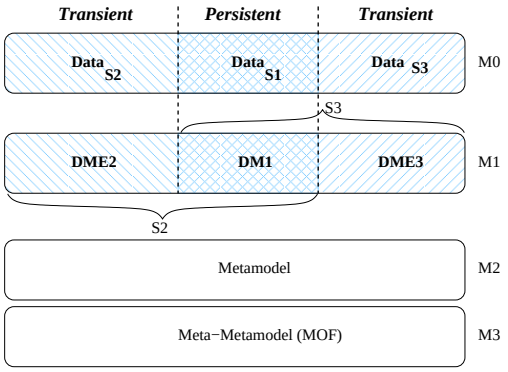


Figure 3. Data model structure

3.1. Transient Model Extension use cases

To get a model for a specific software layer other than the data source layer, its data model normally needs to be extended, which is done on the fly. We call this mechanism Transient Model Extension (TME). It is illustrated by arrows $T1$ and $T2$ between the data structures in Fig. 1 and by fasciated areas in Fig.3.

Like the extensions of a data structure implemented in an object-oriented language follows the rules of the language

compiler, extensions of the model have to be compliant to its metamodel.

Typical TME use cases are:

Attribute Extension. Extending a class with additional attributes is done to model additional information, that belongs to this class and can be computed from other class attributes or associated classes. An extended attribute needs to be of a type already defined in the model.

Additional Associations. Requirements of an upper software layer can require additional associations between classes, that are not associated in the data source. Usually this is done for simplifying associations of related classes in the model. Consider a class A which has a one-to-many association to class B. Class B in turn has a many-to-one association to class C, which has an association with a class D. If now an object of class A needs the associated objects of class D, it has to traverse two associations. This situation can be simplified by directly associationg class A with class D.

Class Extension. Extending the model with additional classes is a combination off all previous use cases. Adding additional classes can be done to meet the requirements for data redundancy or to simplify the functions in the domain layer.

The *data gathering mechanism* for the extended model elements is specified in the TME process. Calculation of model element values can be performed with different mechanisms.

The first approach is the use of a *hook* function, which is called everytime the value of the extended element is requested. The other alternative implements the *observer* pattern [7], implying changes to the extended model element only when one of the observed model elements changes.

One important aspect of the TME approach is the temporary extension of a data model. Because the same data model of the data source layer is used as basis for the domain and presentation layers, extensions applied by these layers should not affect the (data) models of other layers.

3.2. Generic Components and TME

Generic components can be seen as software components for which specific properties have been left variable during components implementation [2]. For AMDD it means that component functionality is based on a data model as well as a metamodel and the model is used for configuration.

In case of generic components TME can be used to configure the runtime behavior of the component. Imagine a user interface component like a table, which dynamically displays data of a given class in the associated data model. Extending this class with additional attributes leads to displaying of additional table columns.

4. Implementational considerations

A more detailed data model of the warehouse management system (WMS) use case presented in Sect. 2 is used for demonstration of the TME mechanism. Although this example has a higher degree of complexity, we note that this example is still a simplified version of our real world application.

Provided that the structure of the persistent data has been modeled, an **Entity Container** (EC) [13] can be used as a model-based object oriented data cache. The architecture of the EC is shown in Fig. 4. The EC provides distinguishable objects called entities, identified by a unique value. It operates on two levels of the four-level metamodel hierarchy of OMG implementing the instance of relation between the two levels.

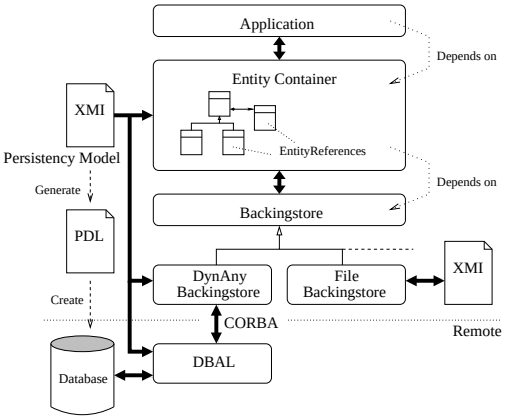


Figure 4. Entity container architecture

The UML data model is stored in a file using the XML Metadata Interchange (XMI) format. This file contains the UML model of the persistent data and is used by the EC and the associated backingstore, such as an object-relational bridge (DBAL). Data entities in the EC are accessed using a dynamic interface.

In our MDD approach the database is created from the persistency model, which is also used to configure the EC and its associated backingstore.

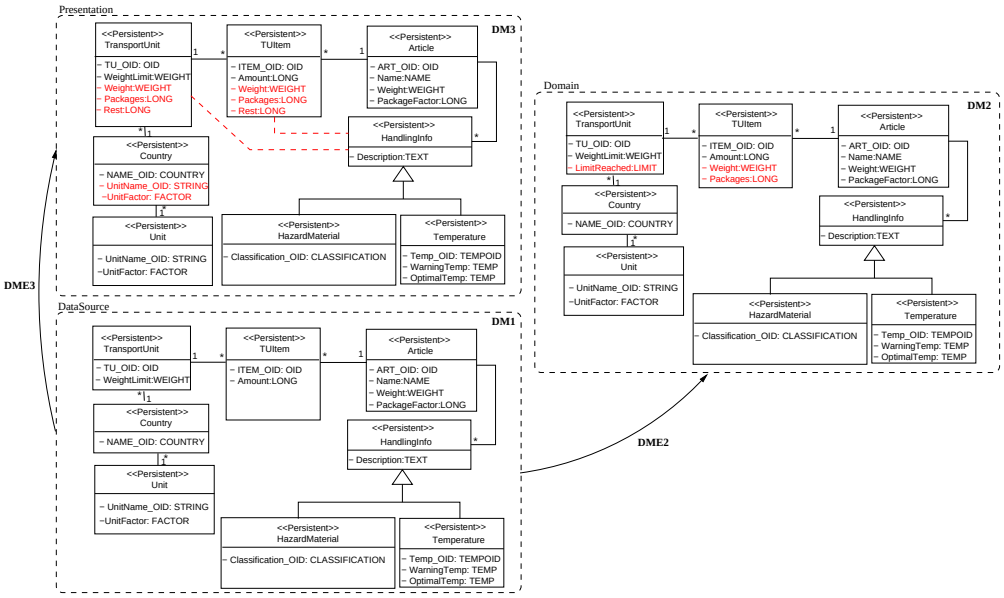


Figure 5. Data models used by software layers for extended example

4.1. Extended example

Instead of the data structure shown in Fig. 2, the detailed data structure allows a mixture of items from different articles in the same box. Each article defines a weight and can be associated with additional handling informations like its preferred temperature or its hazard class. Additionally each box is associated with a target country, which has a preference for a unit system.

Fig. 5 shows the data models of the different structures used in the application. $DM1$ defines the data structure $S1$ of the data source layer, which is used for generating a database by applying the previously described MDD approach. In the persistent data storage only minimal information is saved about the items and boxes, which are modeled as class `TransportUnit`.

The following use cases need to be fulfilled by the different layers of the warehouse management system for goods issue:

UseCase I: It is required to calculate the weight of all items placed on a transport unit. The function can be used to inform a user whether a transport unit is overweight, which can be caused by the individual weight of the different articles.

UseCase II: A user needs a control for displaying the number of items on a transport unit. As stated in section 2 for better usability the displayed item count is split up in packaged items and rest items. Note that every article has an individual factor for calculating the number of packaged items and rest items.

Additionally, a user should also be able to control the weight of the transport unit. The weight is displayed in the preferred unit system of the the inspected transport unit's target country.

The WMS is implemented using the previously described software layers. Because the persistent data model is used for generating the database, it defines the model for the data source layer. All application layers mentioned above are based on the data from the data source of the WMS.

To meet the requirements of the previously defined use cases $DM1$ in Fig. 5 needs to be extended applying the TME mechanism, which is illustrated by the arrows labeled $DME2$ and $DME3$. UseCase I is implemented with the additional attributes contained in $DM2$, while UseCase II is realized by additional attributes and associations in $DM3$.

For $DM2$ the value of the additional weight attribute in `TUItem` is calculated using:

$$Weight_{TUItem} = Amount_{TUItem} * Weight_{Article} \quad (1)$$

The value of this attribute is used in the calculation of the weight attribute of `TransportUnit`, which is based on equation:

$$Weight_{TU} = \sum_{TUItems} Weight_{TUItem} * UnitFactor_{Country} \quad (2)$$

For UseCase II the value of the package and rest attribute of `TUItem` is calculated using the *PackageFactor* attribute of the associated `Article` and the *amount* attribute of `TUItem`.

Keeping these facts in mind the following situation can be thought of. There can exist some articles, which also have a heavy-weight packaging. If this fact needs to be considered in the WMS, the following modifications have to be made.

First a new attribute containing a package weight needs to be added to `TUItem` in $DM1$. Additionally the attribute holding the package count and its corresponding calculation function need to be added to `TUItem` in the TME process. Also Eq. 1 has to be changed to

$$Weight_{TUItem} = Amount_{TUItem} * Weight_{Article} + Packages_{TUItem} * PackageWeight_{TUItem} \quad (3)$$

With this additional extension one benefit of the TME method is demonstrated. Because Eq. 2 relies only on the value of the weight attribute, changes in the Eq. 1 calculating the value of this attribute are transparent to equation Eq. 2.

4.2. TME with the Entity Container

Listing 1 illustrates the code for the data source layer. This layer holds the data model of the application. It also provides access to the data source. According to Fig.4 each EC uses a data model and a backingstore, which holds the connection to the data source. The parameter for initialization of the EC in the upper layers of the application are retrieved from the data layer.

Listing 1. Data source layer

```
1  ..
       // Datastructure holding the data model
3      private IModel dataModel_;

5      // Backingstore connecting to the database
       private IBackingStore bs_;
7
           /**
9          * Public constructor initializing data layer
       **/
11     public DataLayer() {
           // initialize the data model
13         dataModel_ = Model.create4TME("resource/WMSModel.xml.zip");
           // open connection to database for backingstore
15         IPersistency persist = new RemoteDbalPersistency();
                   // initialize backingstore
17         bs_ = new JavaDynAnyBackingStore(dataModel_, persist);
       }
19
21     //Method for retrieving the model of the data source layer
       public IModel getDataModel() {
23         return dataModel_;
       }
25
       // Method for getting access to the data source
27     public IBackingStore getBS() {
           return bs_;
29     }
   ..
```

Listing 2 demonstrates the implementation of the business function for checking the current weight of an item. This method is invoked by providing the ID of the current transport unit. In the initialization phase of this method the data model of the data source layer is retrieved. This model is extended with an additional attribute, containing the weight of the transport unit item. A new EC is instantiated, which is based on the extended model and the backingstore contained in the data source layer.

In this example we demonstrate, that the value for the additional attribute is provided using a hook function.

Listing 2. Implementation of UseCase I

```
2    // business function calculating weight limit
   public boolean weightLimitReached(String tuID){
4    DataLayer dl = Application.getDataLayer();
     // get persistent data model
6    IModel layerModel = dl.getDataModel();
     //apply TME for attribute weight of class TUItem
8    IAttribute weight = layerModel.TME_addNewAttribute(ITEM, attr_Weight ,WEIGHT);
     weight.setHook(new ItemWeightHook());
10   IAttribute limit = layerModel.TME_addNewAttribute(TU, attr_Limit , Limit);
     limit.setHook(new LimitHook());
12
     //initialize new backingstore EC
14   IBackingStore bs = dl.getBS();
     BackingStoreEntityContainer ec_ = new BackingStoreEntityContainer(
16                                          layerModel,bs);
     // load transport units from datasource
18   ec_.load(TransportUnit);

20   //lookup specific transportUnit
     ITemplate template = new Template();
22   template.setAttribute(TU, tuID);
     IEntityReference ref = ec_.findByPrimaryKey(TransportUnit, template);
24
     // return value of the limit attribute for selected transport unit
26   return ref.getAttribute(attr_Limit)
   }
28 ..

30 private class ItemWeightHook extends EntityAttributeHook {
     // called for attribute Weight in class TUItem
32   public IValue getAttribute(IEntityReference entity, String name) {
      double retVal = 0;
34    ICollection<IEntityReference> articles = entity.getConnections(ARTICLE);

36    // calculate weight of associated attributes
      for (IEntityReference article: articles)
38        retVal += ((DoubleValue)article.getAttribute("Weight")).getValue()*
      ((DoubleValue)entity.getAttribute("Amount")).getValue();
40
      return new DoubleValue(retVal);
42   }
   ..
44
46 private class LimitHook extends EntityAttributeHook {
     // calculates the value of attribute limit in TUItem
48   public IValue getAttribute(IEntityReference entity, String name) {
      double currentWeight = 0;
50    ICollection<IEntityReference> items = entity.getConnections(ITEM);

52    for (IEntityReference item: items)
          currentWeight += ((DoubleValue)item.getAttribute("Weight")).getValue();
54    boolean ret = (entity.getAttribute("limit").compareTo(currentWeight) >= 0);

      return new BooleanValue(ret);
56   }
   }
```

Listing 3 shows the implementation of the modified hook method implementing Eq. 3. This implementation can be applied to the weight attribute in Listing 2.

Listing 3. Changed hook implementing Eq.3

```
1  private class ItemWeightHook extends EntityAttributeHook {
3    public IValue getAttribute(IEntityReference entity, String name) {
5      double retVal = 0;
       ICollection<IEntityReference> articles = entity.getConnections(ARTICLE);
7      for (IEntityReference article: articles)
           retVal += ((DoubleValue)article.getAttribute("Weight")).getValue()*
9      ((DoubleValue)entity.getAttribute("Amount")).getValue()+
       ((LongValue)entity.getAttribute("Packages")).getValue()*
11     ((DoubleValue)entity.getAttribute("PackageWeight")).getValue();
       return new DoubleValue(retVal);
13   }
   }
```

5. Discussion

Having seen the TME implementation we are going to summarize the most important points in the concept of Transient Model Extensions.

Model-interpretation technique: In contrast to other approaches, which mainly rely on code generation, our approach is based on model interpretation based on a metamodel. [5]. This allows dynamic generation of classes and objects based on a data model.

Single persistency model: TME relies on a single persistent data model, which represents the persistent data structures. This model is used as basis for each extended data model.

Assuming a three layered software architecture with the data source layer holding the persistent data model, TME can be used to create extended data structures in the domain layer and the presentation layer. These data structures are needed for support of additional attributes or relations according to the use cases implemented in each layer.

Separation of concern: TME can be used for implementing a view on the persistent information stored in the application's data source. This is not only important for implementing different software layers. Another use case is the implementation of different components using the same persistent information.

Code modularization: In a method working with TME first the persistent data model is extended and data gathering methods are added, implementing hooks and observers. In a second step the extended model can be applied to an object oriented data cache. So the return value of the method can be based on the extended attributes. Therefore the method can only contain model extension and model querying statements, while calculation can be performed in small code fragments used by the hooks and oberserver. These hooks and observers can be also shared between different model extensions allowing an improved reusability.

5.1. Related persistent data frameworks

To realize the mapping of persistent data in dynamic data structures several widely known technologies exist.

Microsoft's **ADO.NET** framework [12] contains several classes, which enable the usage of relational databases or XML files as persistent data sources. Access to the database is provided through an instance of a DataReader or a DataAdapter. While the first one is only used for reading data from the database, the second allows also data manipulation independent of the database type. The data adapter is used by a dataset component, which is an in-memory database.

The dataset is used by other application layers, but it does not utilizes a data model. Therefore extending the dataset is done at source code level, instead of the model abstraction level.

Hibernate [8] aims at providing an object/relational bridge. It allows the user to work with object oriented concepts like inheritance and composition as well as with database relational concepts such as usage of a DML like SQL. The mapping between the Java classes and the database structure is done by a XML file.

Because Hibernate works at the source code abstraction level, supporting modelbased extensions is not in the scope of this technology.

The **Eclipse Modeling Framework** (EMF) [3] is an implementation related to the OMG Meta Object Framework (MOF). A EMF model is based on a metamodel called ecore model. This model defines the content of eAttributes and eReferences, which belong to the eClass elements in the model.

EMF models can be built from Java files, XML files or UML models stored in XMI. This models can be used as input for Java source code generators, producing class files with annotated source code. This code can be edited manually to add functionality. EMF supports the serialization of objects contained in the EMF model in XMI files.

The ecore model defines attributes allowing to control which elements of the model can be serialized. The transient flag defines whether the corresponding element can be serialized. The volatile flag is used to signal that the value of this element depends on the value of other model elements.

Our approach differs in the following points from the EMF modeling concept:

1. The first aspect is the support of simultanous but independent extensions of the persistency data model by different methods. In our understanding the same persistent data model is used by all extending models, while the extensions are only visible within a particular scope. In contrast, extensions to EMF models are globally visible.

2. The second aspect is the used concept of model extension. While EMF supports extension of classes with subclassing an existent class in a model, our approach directly adds attributes and references to existing classes. The advantage hereby is the constant class type of the extended class, so no modification of code expecting a particular class type is required.

6. Conclusion

This paper introduced a new method to ease the model driven construction of layered data-intensive software. Applying the concepts of data modeling using traditional concepts, results in one data model for each software layer. This leads to redundant class definition in different models with respect to the data source.

In this cases changing the data model of the data source becomes difficult, because all corresponding classes in the other data models need to be changed. Furthermore each model is driven by various requirements, which leads to a different number of attributes in the equivalent class depending on data model.

To overcome this challenge, data models of different software layers can be received applying additional extensions on the data model from the underlying software layer. The mechanism for application of the model changes on the fly is called Transient Model Extension (TME).

This paper presented TME use cases including the extension of classes, class attributes and associations. These transient extensions are driven by layer specific requirements.

To demonstrate the advantages of this method, we presented an example, considering various requirements of the domain layer and the presentation layer. Finally we provided a comparision of our approach to related technologies dealing with persistent data structures.

7. Acknowledgment

We would like to thank our company, Salomon Automation GmbH, for their grant to support our research.

References

[1] Scott W. Ambler. *Agile Database Techniques*. Wiley Publishing, Inc., 2003.

[2] L. Baum and M. Becker. Generic Components to Foster Reuse. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, pages 266 – 277, 2000.

[3] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *"Eclipse Modeling Framework"*. Addison-Wesley, "August" 2003.

[4] Ahmet Demir. Comparison of model-driven architecture and software factories in the context of model-driven development. In *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*, pages 75–83, Washington, DC, USA, 2006. IEEE Computer Society.

[5] Ulrich W. Eisenecker and Krzysztof Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[6] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesly, 2003.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[8] "Hibernate", "http://www.hibernate.org/5.html". *"Hibernate Reference Documentation"*, "3.2.0ga" edition, 2006.

[9] Stephan J. Mellor, Anthony N. Clark, and Takao Futagami. Model–driven development — guest editor's introduction. *Software, IEEE*, 20(5):14–18, 2003.

[10] OMG. UML Infrastructure, Version 2.0 . Technical Report 2002-09-01, Object Management Group, 2002.

[11] OMG. UML Superstructure, Version 2.0. Technical Report 2002-09-02, Object Management Group, 2002.

[12] David Sceppa. *Microsoft ADO.NET 2.0 Core Reference*. Microsoft Press, 2006.

[13] Gernot Schmoelzer, Stefan Mitterdorfer, Christian Kreiner, Joerg Faschingbauer, Zsolt Kovács, Egon Teiniker, and Reinhold Weiss. The Entity Container — an Object–Oriented and Model–Driven Persistency Cache. In *HICSS'05, Big Island, Hawai'i, USA, Jan. 3-6*, page 277b. IEEE, 2005.

[14] Bran Selic. Model-Driven Development: Its Essence and Opportunities. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006.*, page 7pp. IEEE, April 2006.

Implementing Model-Based Data Structures using Transient Model Extensions

Michael Thonhauser[1,2], Gernot Schmoelzer[2] and Christian Kreiner[1,2]
[1]Institute of Technical Informatics, Graz University of Technology, Austria
[2]Salomon Automation Gmbh, Friesach bei Graz, Austria
Email: michael.thonhauser@tugraz.at, gernot.schmoelzer@salomon.at, kreiner@tugraz.at

***Abstract*— Software is often constructed using a layered approach to encapsulate various functionality in corresponding layers. Individual requirements of each layer demand layer specific data structures. These data structures typically provide redundant information with respect to the data source.**

Providing a Model Driven Software Development approach for creating these data structures leads to overlapping data models, each containing data structures defined by the data source. Because putting all various requirements of the software layers in a single data model can lead to difficulties, each software layer should only extend the basic data source model with its specifically needed model elements.

The approach presented in this paper applies a mechanism for a dynamic extension of a data model. This extension mechanism is used in the implementational activity of a software process, and allows the changing of a model within a local scope. Using this mechanism, a basic data model can be used by every layer, being extended by additional attributes and classes for satisfying layer specific requirements.

***Index Terms*— Model-driven development, Data modeling, Data Intensive Systems, Software layers**

I. Introduction

In the last years their has been a lot of research on model driven development (MDD) [1]–[3], which has lead to different standards like Unified Modeling Language (UML) [4] and approaches like Model Driven Architecture (MDA) [5]. The aim of an MDD approach is the description of software in an abstract way by making use of a model describing the designed software. This model specifies attributes of the software, which are needed in the corresponding activity of a software engineering process.

There exist different software engineering process models, like the waterfall model, evolutionary development, formal systems development or iterative approaches such as the spiral model [6]. All of these process models define some fundamental activities, like software specification, software design and implementation, software validation and software evolution. Each activity can be supported by models. Some modeling standards like UML also provide different views of a model (e.g. class diagram, use case diagram, sequence diagram), which are best suited for different acitivities in a software engineering process.

While MDD aproaches often require to finish the specification of models before the modeled applications are implemented, agile approaches like Agile Model Driven Development (AMDD) [7] focus only on the view of a model currently needed. In an AMDD approach for developing a data-intensive system a data model is the most important type of model created at the beginning of the software development process. Because data-intensive applications are software systems that focus on data processing, data visualization and data storage (such as enterprise resource planning systems, banking applications or logistic systems), a data model contains the description of persistent data structures. These data structures serve as the basis of a data-intensive system. To ease the development of data-intensive applications a layered approach is typically chosen for such architectures [8] consisting of three layers, which are called presentation layer, business function layer and data source layer.

Data structures used by a software layer are described in a data model. Because every layer has different requirements on its data structures, data models of various layers may differ, but there exists a common partial data model of all layer specific data models. This common partial data model can be seen as the application's minimalistic data model.

Supposing that this minimalistic data model contains all data required for persistence, data models of the different layers can be produced by adding layer specific data structures to this model. While the minimalistic data model needs to be defined in the first activity of a software engineering process, layer specific additions can be defined in later activities (e.g. during implementation of the corresponding layer). Some requirements on layer specific data structures are not known in the design phase of a layer, because they follow from implementational considerations. In order to support these requirements by a data model a dynamical model extension mechanism has been proposed in [9]. This mechanism is called Transient Model Extension (TME) to point out, that extensions made to the model are available only in the scope (e.g. layer, class method) where they are defined.

This paper is based on "Model-Based Data Processing with Transient Model Extensions," by M.Thonhauser, G.Schmoelzer and C.Kreiner, which appeared in the Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, Tucson, AZ, USA, March 2007.

For demonstration purposes of the TME mechanism a warehouse management system is introduced in Sect. II giving an example of a layered design for a data-intensive system. Sect. III discusses the dependencies of layered software and data models in more detail, while Sect. IV introduces the TME approach. Sect. V describes an implementation scenario of the examplary WMS using the TME approach.

II. Motivating Example

We begin by introducing a simplified warehouse management system (WMS), which is implemented in the business domain of logistics. The example system supports the process of managing transport unit items (TUItem) being stored in a warehouse. Each TUItem is an instance of a specific article and is contained in a transport unit (TU).

Our warehouse management process uses three stages; at the incoming goods stage TUs are received from suppliers. In the second stage received TUs are stored in a high rack. If a customer orders an article a TU containing a corresponding TUItem is looked up in system. According to the strategy (e.g. best-before date, fastest available TUItem), the found TU is delivered to the third stage of our warehouse, the goods issue. In this stage the ordered TUItems are collected together and stored in another TU. While the filled up TU is delivered to the customer, the original TUs are returned to the high rack stage.

Fig. 1 depicts the modules of the described WMS. Note that each stage of the warehouse management process is associated with one module of the system. Each module is constructed using a layered approach.

The chosen approach consists of three layers. A two layered approach would also be feasible, which would require to split up the functionality of the business layer. This approach is often realized using fourth generation programming languages, like SQL, to realize business functions with stored procedures and aggregate functions. The drawback of this approach is the complexity of the realized queries.

Using a three layered approach queries can be transfered to the business logic layer. They can be split up in programming statements, which rely on smaller queries being responsible for retrieving the required data from the data-source. Often this approach is related to usage of an object-relational mapper in the data source layer, which maps the existing objects onto relational database tables.

Because the majority of currently used programming languages for constructing such systems, such as C#, Java or C++, follows the object oriented paradigm, data structures can be described using a class diagram view of the data model.

III. Related work

A. Layered software

In a layered approach such as proposed in [8] each software layer internally uses data structures, holding

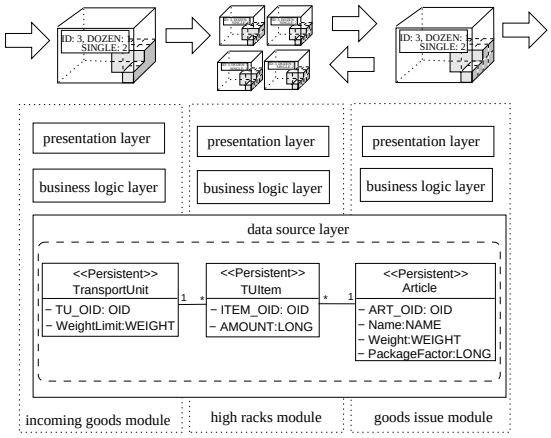


Figure 1. Software layers of warehouse management system

the needed data of this layer. Fig. 2 shows these data structures and depicts the dependencies between them. Every data structure can be seen as a layer specific view on the information generally available to the software. Data structure $S1$ holds the transformed persistent data, which is compliant to the structure of the data source. $S2$ repesents the data structure used by the domain layer and $S3$ is contained in the presentation layer.

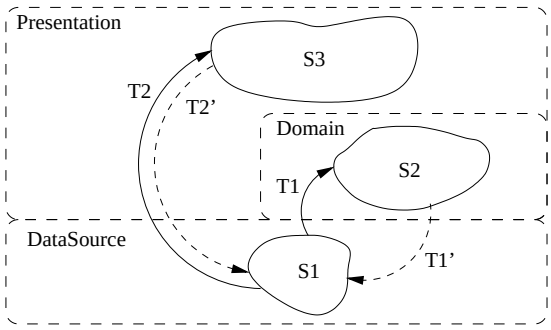


Figure 2. Data structures in software layers

The examplary WMS illustrates different requirements for data structures in various application layers. Requirements for $S1$ are the reduction of redundant information, thus providing support for *data normalization* [7]. This reduction is needed for optimal storage usage, efficient data transfer and to avoid inconistency of stored data. Also the data structure should be easily mappable to the persistent data source.

$S2$ is driven by requirements for *simplifying complexity* of the application logic, adding *transaction handling* and *data consistency checks*. This aims at avoidance of inconsistent data and is done through *adding data redundancy*.

$S3$ contains the data displayed at the presentation layer. Its content meets user requirements and *usability aspects*, which often requires displaying additional information.

As illustraded in Fig. 2 there exist several dependencies

of the data structures in the different layers. Because $S1$ holds all the persistent information available to the application, additional data in $S2$ and $S3$ is normally related to data contained in $S1$. This requirement can be fulfilled by extending $S1$ with a transformation $T1$ or $T2$.

The way this transformation is performed depends on the abstraction level used for application development.

- In a traditional software engineering approach implementation is done by using a relational database and fourth generation programming languages like SQL. This programming language type often includes a database manipulation language (DML) used to define the data structures $S1$, $S2$ and $S3$. In this case the transformation $T1$ and $T2$ of the data structures is done by extending the DML statements.
- A higher level of abstractions is provided by object oriented languages combined with object-relational methods [8]. This solution enables type safety for $S1$, $S2$ and $S3$ consisting of classes. The transformation is done with additional source code for the implementation of $S2$ and $S3$.
- A further abstraction level is reached by using a MDD approach. This leads to a corresponding data model $DM1$ for $S1$, $DM2$ for $S2$ and $DM3$ for $S3$. Following the dependencies of the different data structures, the dependencies of the data models can be seen as $DM1 \subseteq DM2$ and $DM1 \subseteq DM3$. In this case transformation $T1$ or $T2$ is done by extending $DM1$. This extension adds information about additional data needed in the presentation layer leading to $DM3$ and to $DM2$ respectivly in the domain layer.

  Note that this mechanism is called Transient Model Extension (TME), because the extensions to $DM1$ are only visible at the scope where they are applied. There exist also many small extensions for $T1$ and $T2$. Since $DM1$ is used as the basic model for this mechanism, $DM2$ and $DM3$ does not need to be stored in the corresponding layers. Only the transformation rules needed for construction of the appropriate data model need to be saved instead.

B. Model Driven Development

Model Driven Development (MDD) [1] is an approach to implement a software system by describing it with a Platform Independent Model (PIM). A PIM defines associations between data and behavior of the software and it is used as input for generators producing a platform specific model (PSM). To support MDD the Object Management Group (OMG) has released the Model-Driven Architecture (MDA) containing standards, which enable specification and transformation of models. Another approach to MDD are Software Factories, which are proposed by Microsoft and can be seen as a new software development paradigm. Differences of these two approaches are discussed in [10].

Models of software design are often specified using Unified Modeling Language (UML) [11], another standard of the OMG. UML models are based on a metamodel and are situated in the user model layer of the four-level metamodel hierarchy [4]. UML describes several diagrams, which can be used to model different aspects of a software. Structural and behavioral diagrams are differentiated. One example of a structural diagram is the class diagram. It is used to model the structure of classes, such as attributes and methods, as well as associations between different classes in the model.

For data modeling purposes the metamodel of a class diagram can be extended focusing only on class attributes and associations [7].

Agile Data Modeling relies on iterative construction of data models, where each data model satisfies the requirements needed in the current iteration. It is best suitable for applications that rely on relational databases for persistent data storage. Agile Model Driven Development (AMDD) also uses an iterative approach, instead of extensive models being generated in the regular MDD process.

C. Related persistent data frameworks

To realize the mapping of persistent data in dynamic data structures several widely known technologies exist.

Microsoft's **ADO.NET** framework [12] contains several classes, which enable the usage of relational databases or XML files as persistent data sources. Access to the database is provided through an instance of a DataReader or a DataAdapter. While the first one is only used for reading data from a database, the second allows also data manipulation independent of the database type. The data adapter is used by a dataset component, which is an in-memory database.

The dataset is used by other application layers, but it does not utilize a data model. Therefore extending the dataset is done at source code level instead of the model abstraction level.

Hibernate [13] aims at providing an object/relational bridge. It allows the user to work with object oriented concepts like inheritance and composition as well as with database relational concepts such as usage of a DML like SQL. The mapping between the Java classes and the database structure is done by a XML file.

Because Hibernate works at the source code abstraction level, supporting model based extensions is not in the scope of this technology.

The **Eclipse Modeling Framework** (EMF) [14] is an implementation related to the OMG Meta Object Framework (MOF). A EMF model is based on a metamodel called ecore model. This model defines the content of eAttributes and eReferences, which belong to the eClass elements in the model.

EMF models can be built from Java files, XML files or UML models stored in XMI. These models can be used as input for Java source code generators, producing class files with annotated source code. This code can be edited manually to add functionality. EMF supports the

serialization of objects in XMI files, if they are based on an EMF model.

The ecore model defines attributes allowing to control which elements of the model can be serialized. The transient flag defines whether the corresponding element can be serialized. The volatile flag is used to signal that the value of this element depends on the value of other model elements.

Our approach differs in the following points from the EMF modeling concept:

1) The first aspect is the support of simultanous but independent extensions of the persistency data model by different methods. In our understanding the same persistent data model is used by all extending models, while the extensions are only visible within a particular scope. In contrast, extensions to EMF models are globally visible.
2) The second aspect is the used concept of model extension. While EMF supports extension of classes with subclassing an existent class in a model, our approach directly adds attributes and references to existing classes. The advantage hereby is the constant class type of the extended class, so no modification of code expecting a particular class type is required.

D. Partial model techniques

While many model based applications rely on large monolithic models, there exist alternatives in the domain of Domain Specifc Languages (DSL) [15], which rely on the management of multiple partial models. These partial models can be linked together to create one application specific model or can be used for partial source code generation, which can be linked.

Another approach based on metamodels is descriped in [16]. This approach uses core models and fragment models, which conform to the same metamodel. These models are then linked together following a formal definition, which ensures that the resulting model is also conform to the metamodel.

In our approach of transient model extension used models are also expected to be conform to their metamodels. But instead of linking these partial models, our approach allows the specification of model extensions for a specific model in other forms, e.g. in a programming language. Another distinction is the corresponding acitivity in the software process used for creating the extension of models or performing the linkage of partial models. Often partial models are defined during design activities and are often linked before starting with implementational activities in the software development process. This can be done for enabling generators to produce corresponding application artifacts (like source code or database setup scripts) [17]. In contrast the TME approach is applied during implementational activities of a software development process to dynamically manipulate the model, where the manipulations have a local scope. Looking at the different roles in a software process, model linking approaches are applied by application and database designers, while our approach is used for supporting the software developer in implementing the layers.

IV. Transient Model Extension

The structure of a data model can be based on the four-level metamodel hierarchy of the OMG, where level M3 (meta-metamodel) is the same for each data model. Level M2 contains the general metamodel and additional metamodel extensions, allowing the specification of domain specific data models in M1. These domain specific data models maintain the domain specific classes and associations being used for construction of objects which correspond to model level M0. Generally speaking, every model can be seen as an instance of the model in the lower layer. Considering this fact, it is obvious that changing the model in level M1 results in additional data contained in level M0.

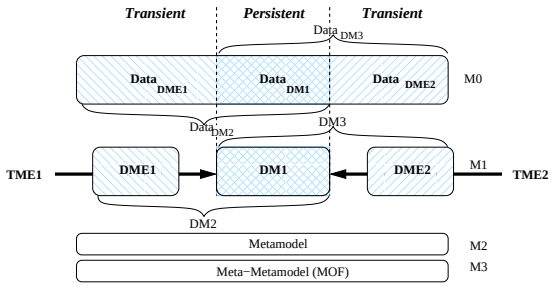


Figure 3. Data model structure

Fig. 3 illustrates the elements needed in the four-level metamodel hierarchy for realization of the data structures defined in Fig. 2. According to this figure $DM1$ is part of every data model defined in level M1. $DM1$ defines the data structure $S1$, which contains $Data_{DM1}$ in M0. In Fig. 2 mappings $T1$ and $T2$ are defined for creating the corresponding data structures $S2$ and $S3$. Mapping $T1$ is represented by $TME1$ and $T2$ is represented by $TME2$ respectivly. Each of these mappings defines a Data model extension (DME) of M1.

According to the assumptions made in Sect. III-A, $DM2$ is equivalent to $S2$ and is defined as a combination of $DM1$ and $DME1$. In the same way the data model $DM3$ results from combining $DM1$ with $DME2$. Every model has corresponding data structures and data in M0.

As stated in Sect. III-A, $DM1$ is used by the data source layer, $DM2$ by the business function layer, and $DM3$ by the presentation layer.

Applying the AMDD approach to a layered software architecture results in the definition of a data model containing the data structures used by the data source layer. Based on the persistent data model the corresponding data models for the domain and the presentation layer are defined. According to Fig. 3 a separate definition of each data model used by the different software layers is not feasible. Common data definitions needed by different

layers are duplicated in the models, leading to maintenance difficulties of the shared information. Also each layer needs to store its own data model, which can lead to increasing storage requirements by the application. To overcome these problems a single data model is defined in the data source layer, containing the persistent data structures.

To get a model for a software layer other than the data source layer, its data model normally can be extended by a mechanism called Transient Model Extension (TME). It is illustrated by arrows $T1$ and $T2$ between the data structures in Fig. 2 and $TME1$ and $TME2$ in Fig.3 respectivly.

Note that the extensions provided to the data model can be applied for different scopes. As stated above, an extension of the model can be defined and used by all classes being part of a layer. Another possibility is illustrated in Fig. 4; the business logic layer contains two methods, which apply a TME to the data model referenced from the data source layer. In this case each TME is only visible to the defining method.

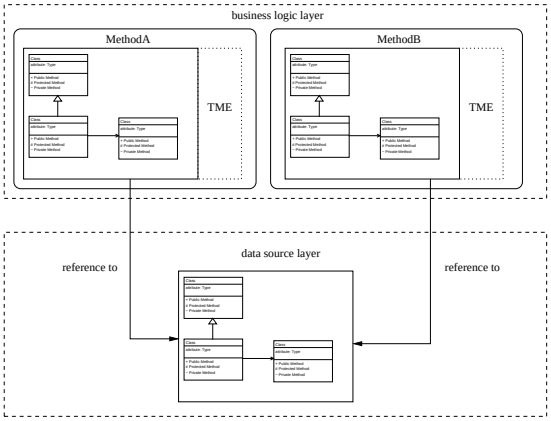


Figure 4. TME with method scope

A. TME types

As extensions of data structures implemented in an object-oriented language need to follow the rules of the language compiler, extensions of the model have to be compliant to its metamodel. According to this fact, an extension of the model element needs to be an instance of an element defined in the metamodel.

The following elements of a class metamodel are used in a TME mechanism for data-models.

1) Attributes: Extending a class with additional attributes is done to model additional information, that belongs to this class. Often this information can be derived from other class attributes or (attributes of) associated classes. The type of an extended attribute has to be defined in the model before the extended attribute is defined.

Fig. 5 demonstrates an example of a TME with an attribute for the datamodel of our WMS defined in Sect. II. For retrieving the weight of a TU the sum of the weight of all TUItems has to be calculated. The weight of a TUItem again depends on the weight of the corresponding article and the amount of items located in the TU. Dynamically adding the weight attribute to the classes *TransportUnit* and *TUItem* allows to divide the calculation descriped above. Also the result of this calculation has the same metainformation as the weight attribute of the article.

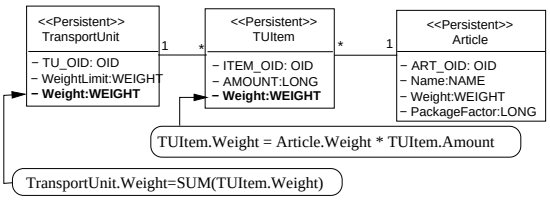


Figure 5. TME for weight attributes

2) Associations: Business functions may require additional associations between classes, that are not associated in the data source. Usually this is done for simplifying associations of related classes in the model. Again this TME type is demonstrated using the datamodel of the WMS. Consider the requirement to display information on all articles contained in a TU. This requirement can be fulfilled by extending the datamodel of the WMS with a 1:n association between the class *TransportUnit* and the class *Article*.

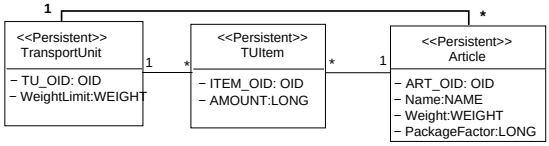


Figure 6. TME for associating articles with TUs

3) Classes: Extension of the datamodel with new classes requires at least an additional TME with a class attribute. It can also require a TME with an association, to connect the new class to an existing class in the model.

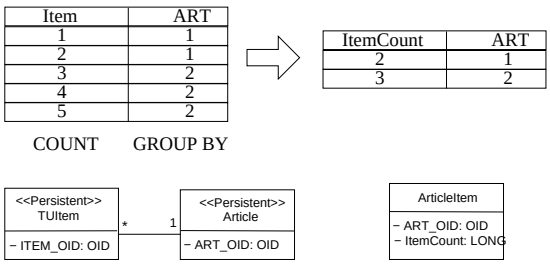


Figure 7. TME for a class

Fig. 7 shows a use case, which does not require a TME with an association. This example is driven by the requirement to count the number of TUItems for each article. Looking at the database tables shown in

the upper section of the figure, it is clear, that this requirement can be solved, by applying a query containing a GROUP BY clause and a COUNT function on the TUItem field. This requirement is solved by a TME with the class *ArticleItem*, which is used for constructing the entity objects containing the result from the database query. Using an additional class satisfies the following constraints which arise from the database query:

COUNT(x) → int

GROUP BY (x) → x

Every entity of the class *ArticleItem* is therefore fully defined within the metamodel, i.e. every attribute contains its correct metainformation. Note that this requirement could also have been solved by adding an additional attribute to the class *Article* instead of specifying a totally new class. The reason for using a class extension are performance considerations, which we will discuss in the next section.

B. Clientside and serverside TME

The initial idea of the TME mechanism has been the client side extension of the datamodel, e.g. to enable additional columns in table widgets being shown in the presentation layer. But in fact most data-intensive systems are build using a distributed architecture. In this context the meaning of software layers is equivalent to the meaning of software tiers. A tier is a layer, which is not deployed on the same machine as the other layers in the system. In a three tier system two configurations are usually deployed. The first configuration, also known as fat client, requires presentation tier and business logic tier being installed on the same machine. The second configuration, known as thin client, requires only the presentation layer being deployed on the client machine, while business logic tier and data source tier are deployed on the server.

Both configurations think of the data source tier being deployed on a server machine. This server contains the datasource, which is often a relational database, or it has a high performance network connection to the machine containing the datasource. In the second situation performance considerations can become important for applying TME to the datamodel.

The example demonstrated in Fig. 7 can also be implemented using a TME with an attribute for the *Article* class. But by applying the TME with an attribute the software engineer requires the infrastructure to transfer first all *TUItem* objects and all *Article* objects from the datasource layer to the layer containing the extended datamodel. Having finished the transfer the sum up of the corresponding TUItem entities for each article can be performed. This variant requires a big amount of data to be transferred, while the example in Fig.7 makes use of the advantages of the RDBMS, applying the corresponding query in a DML and then constructing new entities of the class ArticleItem from the returning set of this query.

C. Other design considerations

Beside the descision to use client or serverside TME another design consideration is the *data gathering mechanism* for the extended model elements. Especially in the case of attribute TME it is always required to specify a mechanism for retrieving the value of an attribute.

The first approach is the use of a *hook* function, which is called everytime the value of the extended element is requested. The other alternative implements the *observer* pattern [18], implying changes to the extended model element only when one of the observed model elements changes.

For demonstration purposes consider the example shown in Fig. 5. The example used two TMEs adding a weight attribute to the *TUItem* and *TransportUnit* class. Considering the requirement, that the amount stored in *TUItem* and the weight of the article are fixed, usage of two hook methods each implementing one of the equations shown in Fig. 5 is enough. If this requirement changes, additional observer methods need to be specified for the corresponding variable attributes. These observer methods need to trigger the two hook functions for updating the additional attributes.

Another important aspect of the TME approach is the temporary extension of a data model. Because the same data model of the data source layer is used as basis for the domain and presentation layers, extensions applied within these layers should not affect the (data) models of other layers.

D. Generic Components and TME

Generic components can be seen as software components for which specific properties have been left variable during components implementation [19]. For AMDD it means that component functionality is based on a data model as well as a metamodel and the model is used for configuration.

In case of generic components TME can be used to configure the runtime behavior of the component. Imagine a user interface component like a table, which dynamically displays data of a given class in the associated data model. Extending this class with additional attributes leads to displaying of additional table columns.

V. Implementational considerations

For demonstration purposes the uses cases presented in Sect. IV-A are implemented using the layered WMS introduced in Sect. II. The examples are realized using the framework of our real world application.

Provided that the structure of the persistent data has been modeled, an **Entity Container** (EC) [20] can be used as a model-based object oriented data cache. The architecture of the EC is shown in Fig. 8. The EC provides distinguishable objects called entities, identified by a unique value. It operates on two levels of the four-level metamodel hierarchy of OMG implementing the instance of relation between the two levels.

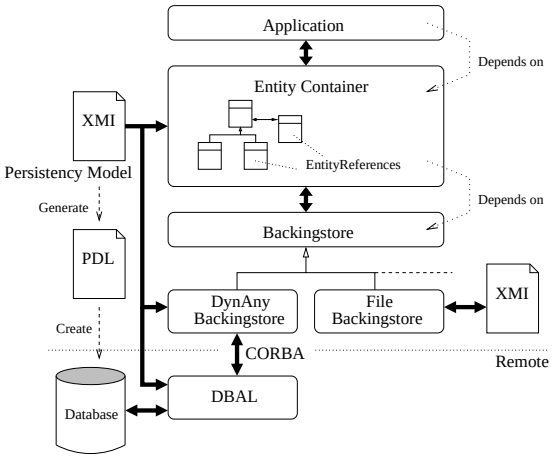


Figure 8. Entity container architecture

The UML data model is stored in a file using the XML Metadata Interchange (XMI) format. This file contains the UML model of the persistent data and is used by the EC and the associated backingstore, such as an object-relational bridge (DBAL). Data entities in the EC are accessed using a dynamic interface.

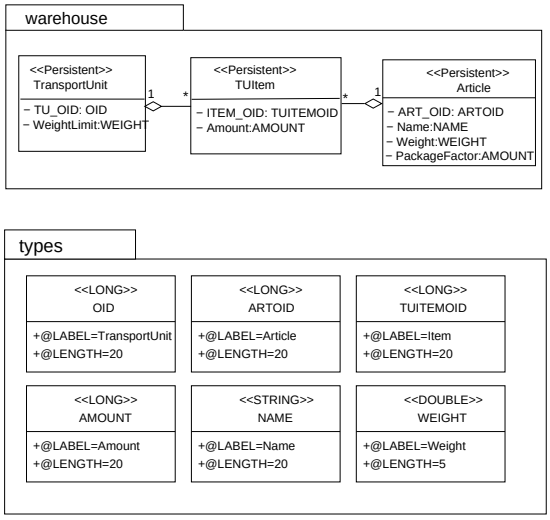


Figure 9. Case study data model

In our MDD approach the database is created from the persistency model, which is also used to configure the EC and its associated backingstore. Fig. 9 shows the data model used for our examples. It consists of two packages, with the warehouse package containing the persistent data classes. Attributes of these classes use the type classes defined in the types package. This data model is applied for creating a database containing the data shown in Table I. There exist two TUs, with one containing two TUItems and the other being empty. The two TUItems are instances of different articles. These data are used for demonstrating

TABLE I.
DATA USED FOR TABLE IN FIG. 10 AND FIG. 11

TU	TUItem	Amount	Article	Article.Weight
1234	211	2	815	3.0
1234	222	1	816	4.0
1235	-	-	-	-

the following use cases:

1) Applying TME for weight attributes (see Fig. 5)
2) Applying TME for a new class (see Fig. 7)

A. TME with the Entity Container

The EC framework is currently implemented for C++ and Java. Because both versions provide the same API, the following examples demonstrated for Java can also be applied in C++. While the C++ implementation is packaged in dynamic libraries, the Java implementation is deployed using the Eclipse plugin mechanism.

Eclipse is an open source Integrated Development Environment (IDE) written in Java, which has been initiated by IBM. The Eclipse IDE is based on the Open Service Gateway Infrastructure (OSGI) [21] framework and makes use of different design patterns [22]. It offers support for development of modularized applications consisting of several plugins. Each plugin contains classes and development fragments belonging together, and is used for realising a well-defined use case. A plugin manifest defines the plugin configuration data and contains extension point (EP) definitions, enabling the dynamical extension of plugin behaviour at runtime.

Listing 1. TME for weight attribute using sourcecode

```
String TU = "::warehouse::TransportUnit";
String TUItem = "::warehouse::TUItem";
String typeWeight="::types::Weight";

public EntityModel getModel() {
        EntityModel persistModel = datamodel.Activator.getDefault().getModel();
        try {
      IAttribute attr = persistModel.TME_addNewAttribute(TUItem,
                                                         "Weight",
                                                         typeWeight);
      attr.setHook(new TUItemWeightHook());
      attr = persistModel.TME_addNewAttribute(TU,"Weight",typeWeight);
      attr.setHook(new TUWeightHook());
        }
        catch (Exception ex) {
          System.out.println(ex.getMessage());
        }
    return persistModel;
}
```

TME use cases are typically realized as statements in the code of the programming language (see Listing 1 for an example realizing usecase (1) defined in the previous section).

Listing 2. EntityAttributeHook for TransportUnit weight attribute

```
public class TUWeightHook extends EntityAttributeHook {

  public IValue getAttribute(IEntityReference entity, String name) throws
                                                    EntityAttributeHookException
  {
    double weight = 0.0;
    try {
      for (IEntityReference ref :entity.getConnections("TUItem")) {
        weight += ((DoubleValue)ref.getAttribute("Weight")).getValue();
      }
    }
    catch (Exception ex) {
      System.out.println("Error calculating weight for '" + name +"':"+
                                                          ex.getMessage());
    }
    return new DoubleBuilder(weight).newValue();
  }
}
```

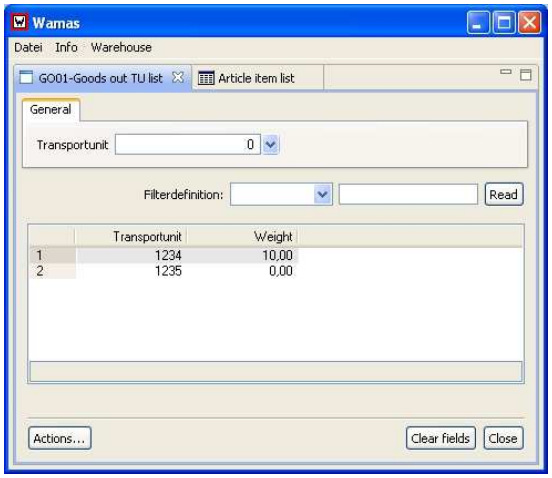


Figure 10. Dialog displaying extended *TransportUnit*

Listing 3. Table configuration for Fig. 10

```
// retrieve table from dialog layout
table_ = (TableView) swtLayout.getControl("ViewPanel.TableView");
//retrieve extended model
EntityModel model = userinterface.Activator.getDefault().getModel();
//construct Entitycontainer
IEntityContainer ec = ContainerFactory.createContainer();
// construct table configuration
TableViewConfiguration config = new TableViewConfiguration(ec,
                                                           model,
                                "::warehouse::TransportUnit#ENTRY");
// configure table
table_.setTableViewConfiguration(config);
```

Listing 2 presents a class, which is derived from the abstract class *EntityAttributeHook*. The hook method **getAttribute()** receives the extended entity of type *TransportUnit* and sums up the weight attributes of its connected *TUItem*s. Note that an object of this class is set as the hook object for the extended attribute weight in Listing 1. The second hook class *TUItemWeightHook* is also derived from *EntityAttribueHook* and is providing the same method getAttribute(), which calculates the weight of a *TUItem* by multiplying the weight of the associated *Article* with the amount attribute of *TUItem*.

A screenshot of a dialog displaying a table, which is configured with the code presented in Listing 3 is shown in Fig. 10. The dialog consists of a filter panel, containing a combo box with all *TransportUnit*.TU attribute values, and the table mentioned above, which displays the extended class *TransportUnit*.

The value of the weight column is calculated using the hook function defined in Listing 2 and a corresponding hook function for the weight attribute in TUItem. The data displayed in the table widget are based on the data defined in Table I.

Listing 4 illustrates the code for the data source layer. This layer holds the data model of the application. It also provides access to the data source.

According to Fig.8 each EC uses a data model and a backingstore, which holds the connection to the data source. The parameter for initialization of the EC in the upper layers of the application are retrieved from the data source layer.

Listing 4. Data source layer

```
..
    // Datastructure holding the data model
    private IModel dataModel_;

    // Backingstore connecting to the database
    private IBackingStore bs_;

    /**
     * Public constructor initializing data layer
    **/
    public DataLayer() {
        // initialize the data model
        dataModel_ = Model.create4TME("resource/WMSModel.xml.zip");
        // open connection to database for backingstore
        IPersistency persist = new RemoteDbalPersistency();
                // initialize backingstore
        bs_ = new JavaDynAnyBackingStore(dataModel_, persist);
    }

    //Method for retrieving the model of the data source layer
    public IModel getDataModel() {
        return dataModel_;
    }

    // Method for getting access to the data source
    public IBackingStore getBS() {
        return bs_;
    }
..
```

Usecase (2) from the previous section is realized by a class *ArticleItemCounter* in the business logic layer. This class contains the method **countItems()**, which is presented in Listing 5.

Listing 5. Counting items of an article

```
public static ArticleItemCounterResult countItems() {

String article = "::warehouse::Article";

// retrieving model from data source layer
EntityModel model = DataSourceLayer.getModel();

// extending model with class and attributes
try {
  IPersistent articleItem = model.TME_addNewClass("::warehouse", "ArticleItem");
  IAttribute oid = model.findPersistent(article).lookupAttribute("Article");
  articleItem.addAttribute(oid);
  IPrimitive amount = model.findPrimitiveType("::types::AMOUNT");
  IAttribute itemCount = model.newAttribute("ItemCount", false, amount);
  articleItem.addAttribute(itemCount);
}
catch (WXException ex) {
  logger_.error("Error applying TME "+ ex.getMessage());
}

// constructing EntityContainer with extended model
IEntityContainer ec = ContainerFactory.getInstance().createContainer(model);

// counting the items for each article
try {
  for (IEntityReference articleRef :ec.getAll(article)) {
    long itemCount = 0;
    for (IEntityReference itemRef:articleRef.getConnections("Item")) {
      itemCount += ((LongValue)itemRef.getAttribute("Amount")).getValue();
    }
    LongBuilder itemCountValue = new LongBuilder(itemCount);
    ITemplate templ = new Template();
    templ.setAttribute("Article", articleRef.getAttribute("Article"));
    templ.setAttribute("ItemCount", itemCountValue.newValue());
    ec.createByTemplate("::warehouse::ArticleItem", templ);
        }
        return new ArticleItemCounterResult(ec,model);
  }
  catch (Exception ex) {
    logger_.error("Error calculating itemcount: "+ ex.getMessage());
  }
  return null;
}
```

This method retrieves the data model from the data source layer and applies a TME with the class *ArticleItem*, containing the attributes Article and ItemCount. After applying TME, an EC is constructed for the extended model and is loaded with entitities of the class *Article* and associated *TUItem* entitites contained in the database. For every *Article* in the EC the value of the amount attribute of its associated *TUItem* entitities is summed up, and a new entity of type *ArticleItem* is constructed for the current Article attribute and the received sum of items.

Having counted all items, the EC is returned to the calling class. Listing 6 contains the configuration for the

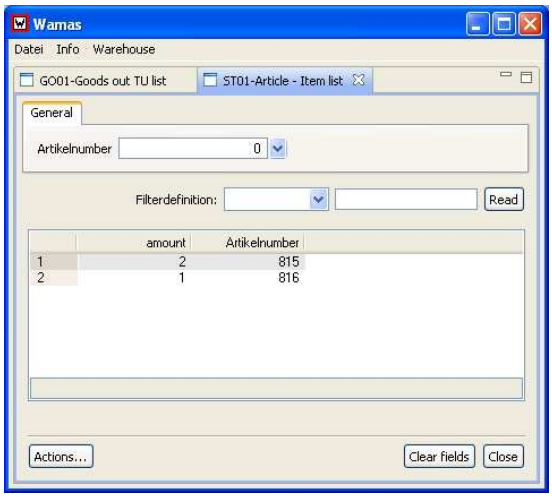


Figure 11. Dialog displaying the number of items for each article

table widget presented in Fig. 11. The configuration is based on the result of the method defined in Listing 5.

Listing 6. Table configuration for Fig. 11

```
// retrieve table from dialog layout
table_ = (TableView) swtLayout.getControl("ViewPanel.TableView");

//call business logic
ArticleItemCounterResult result = ArticleItemCounter.countItems();
TableViewConfiguration config = new TableViewConfiguration(
        result.getEC(), result.getModel(),
"::warehouse::ArticleItem#ENTRY");

// provide configuration
table_.setTableViewConfiguration(config);
```

B. Generic programming using Eclipse plugins

In case of programming with Java and using the Eclipse plugin mechanism, the declaration of TME use cases can be done by contributing to a specific EP, which is defined in the plugin containing the data model.

Listing 7. EP definition for TME with attribute

```
<?xml version='1.0' encoding='UTF-8'?>
<schema targetNamespace="DataModel">
  <annotation>
    <appInfo>
      <meta.schema plugin="DataModel" id="TMEAttribute" name="TME"/>
    </appInfo>
    <documentation>Transient model extension with attribute</documentation>
  </annotation>

  <element name="extension">
    <complexType>
      <sequence minOccurs="1" maxOccurs="unbounded">
        <element ref="TME_Attribute" minOccurs="1" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="point" type="string" use="required">[..]</attribute>
      <attribute name="id" type="string">[..]</attribute>
      <attribute name="name" type="string">[..]</attribute>
    </complexType>
  </element>

  <element name="TME_Attribute">
    <complexType>
      <attribute name="class" type="string" use="required">[..]</attribute>
      <attribute name="attribute" type="string" use="required">[..]</attribute>
      <attribute name="type" type="string" use="required">[..]</attribute>
      <attribute name="AttributeHook" type="string">
        <annotation>
          [..]
          <appInfo>
            <meta.attribute kind="java"
                    basedOn="wx.datamodel.EntityAttributeHook"/>
          </appInfo>
        </annotation>
      </attribute>
    </complexType>
  </element>
</schema>
```

An exemplary definition is shown in Listing 7. This definition is contained in the data model plugin and is used by the extension defined in Listing 8, which is located in the corresponding layer plugin applying the TME mechanism.

Listing 8. TME for weight attribute using EP from Listing 7

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
   <extension
         point="DataModel.TMEAttribute">
      <TME_Attribute
            AttributeHook="TUWeightHook"
            attribute="Weight"
            class="TransportUnit"
            type="::types::Weight"/>
   </extension>

</plugin>
```

The extensions of the data model plugin can be processed each time, the data model is requested. Before returning a new instance of the data model the plugin processes the provided EP contributions, which leads to the data model with TME applied.

Note that both TME attribute use cases, the sourcecode mechanism(Listing 1) as well as the EP mechanism (Listing 8), make use of the same hook class defined in Listing 2.

VI. Conclusion

This paper introduced a new method to ease the model driven construction of layered data-intensive software. Applying the concepts of data modeling using traditional approaches results in one data model for each software layer. This leads to redundant class definition in different models with respect to the data source. Changing the data model of the data source becomes difficult, because all corresponding classes in the other data models need to be changed as well. Furthermore each model is driven by various requirements, which leads to a different number of attributes in the equivalent class depending on data model.

To overcome this challenge, data models of different software layers can be received applying additional extensions on the data model from the underlying software layer. The mechanism for applying these model changes on the fly is called Transient Model Extension (TME). This paper presented TME use cases including the extension of classes, class attributes and associations. These transient extensions are driven by layer specific requirements and are applied with different scopes (e.g. layer specific TME, method specific TME).

To demonstrate the advantages of this method, we presented an example using the framework of our real world application, considering various requirements of the business logic layer and the presentation layer. We also provided a comparision of our approach to related technologies dealing with persistent data structures.

We have found this approach to be useful in constructing large data-intensive systems in the business domain of logistic, because it supports the modularization of different methods in specified data retrieving classes. Because changes to the model are transparent to the

persistent data structures only small changes are needed in the configuration of data displaying widgets. The TME method is also an important part of our approach for implementing a model based software product line for data-intensive systems [23].

ACKNOWLEDGMENT

We would like to thank our company, Salomon Automation GmbH, for their grant to support our research. We also thank Zsolt Kovacs for the interesting discussions, and Carlo Jenetten and Robert Lechner for their support in coding the mentioned tools.

REFERENCES

[1] S. J. Mellor, A. N. Clark, and T. Futagami, "Model–driven development — guest editor's introduction," *Software, IEEE*, vol. 20, no. 5, pp. 14–18, 2003.

[2] B. Selic, "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, no. 5, pp. 19–25, 2003.

[3] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, p. 25, 2006.

[4] OMG, "UML Infrastructure, Version 2.0 ," Object Management Group, Tech. Rep. 2002-09-01, 2002.

[5] D. S. Frankel, *Model Driven Architecture, Applying MDA to Enterprise Computing*. John Wiley & Sons, 2003.

[6] I. Sommerville, *Software engineering (6th ed.)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2001.

[7] S. W. Ambler, *Agile Database Techniques*. Wiley Publishing, Inc., 2003.

[8] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesly, 2003.

[9] M. Thonhauser, G. Schmoelzer, and C. Kreiner, "Model-based Data Processing with Transient Model Extensions," in *ECBS*, 2007, pp. 299–306.

[10] A. Demir, "Comparison of model-driven architecture and software factories in the context of model-driven development," in *MBD-MOMPES '06: Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 75–83.

[11] OMG, "UML Superstructure, Version 2.0," Object Management Group, Tech. Rep. 2002-09-02, 2002.

[12] D. Sceppa, *Microsoft ADO.NET 2.0 Core Reference*. Microsoft Press, 2006.

[13] *"Hibernate Reference Documentation"*, "3.2.0ga" ed., "Hibernate", "http://www.hibernate.org/5.html", 2006.

[14] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose, *"Eclipse Modeling Framework"*. Addison-Wesley, "August" 2003.

[15] J. B. Warmer and A. G. Kleppe, "Building a flexible software factory using partial domain specific models," in *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*. Jyvaskyla: University of Jyvaskyla, October 2006, pp. 15–22.

[16] M. Barbero, F. Jouault, J. Gray, and J. Bézivin, "A practical approach to model extension." in *ECMDA-FA*, ser. Lecture Notes in Computer Science, D. H. Akehurst, R. Vogel, and R. F. Paige, Eds., vol. 4530. Springer, 2007, pp. 32–42.

[17] S. Mitterdorfer, E. Teiniker, C. Kreiner, Z. Kovács, and R. Weiss, "XMI based Model Linking," in *CAINE 2002, International Conference on Computer Applications in Industry and Engineering, Las Vegas, Nevada USA, Nov. 11-13*, E. Nygard, Ed. Cary, NC: ISCA, 2003, pp. 322–325.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Reading, MA: Addison Wesley, 1995.

[19] L. Baum and M. Becker, "Generic Components to Foster Reuse," in *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems*, 2000, pp. 266 – 277.

[20] G. Schmoelzer, S. Mitterdorfer, C. Kreiner, J. Faschingbauer, Z. Kovács, E. Teiniker, and R. Weiss, "The Entity Container — an Object–Oriented and Model–Driven Persistency Cache," in *HICSS'05, Big Island, Hawai'i, USA, Jan. 3-6*. IEEE, 2005, p. 277b.

[21] T. O. Alliance, *OSGi Service Platform Core Specification*, version 4.1, release 4 ed., April 2007.

[22] E. Gamma and K. Beck, *Contributing to Eclipse. Principles, Patterns, and Plugins.: Principles, Patterns and Plugins*. Addison-Wesley Professional, 2003.

[23] G. Schmoelzer, C. Kreiner, and M. Thonhauser, "Platform design for software product lines of data-intensive systems," in *Proceedings of the 33th EUROMICRO Conference*, August 2007.

Michael Thonhauser is currently a Ph.D. student at Graz, University of Technology, Austria and is also working at the research and development department of the company Salomon Automation GmbH. He received his MS degree in informatics and eletrical engineering from Graz, University of Technology in 2005. His research interests include software engineering methods and distributed systems.

Gernot Schmölzer is currently working at the research department for Salomon Automation GmbH, a large logistics software vendor in Europe. He received his Ph.D. in informatics and electrical engineering from Graz University of Technology in 2007. His research interests include model-driven development, software product line engineering and software engineering practices.

Christian Kreiner graduated and received Ph.D. degree in Electrical Engineering from Graz University of Technology in 1991 and 1999 respectively. His research interests include software technology, software engineering and quality management. Christian Kreiner is currently head of the R&D department at Salomon Automation, Austria, focusing on software development for AS/RS (automatic storage/retrieval systems) and researcher at the Institute for Technical Informatics of Graz University of Technology.

Towards a Generic Model Interpretation Runtime Architecture

Michael Thonhauser[1], Christian Kreiner[1]
[1]Institute for Technical Informatics
Graz University of Technology, Austria
Inffeldgasse 16, A-8010 Graz
michael.thonhauser@TUGraz.at

1 Motivation

Today the number of mobile devices is steadily increasing, allowing new kinds of applications for supporting pervasive business workflows. During the execution of such workflows, the pervasive application is applied by different users on different mobile devices.

Several approaches for developing such applications with multiple heterogeneous target platforms have been proposed in the last decade. One of the most prominent approaches is Model Driven Software Engineering, where a platform independent model is used for abstract specification of the application and platform dependent artifacts are generated at development time.

Approaches supporting the assembling and deployment of such applications are often based on Component Based Software Engineering (CBSE) techniques, allowing the specification of components with clear interfaces. Choosing the right interface granularity and making interfaces robust against domain specific changes are important challenges to foster the reuse of a component. An approach to solve this problem has been introduced by the Business Component Factory [1], enabling the specification and composition of components at various levels of detail. Components built with this approach are not directly connected through interfaces, but are plugged in a component execution environment, which resolves the component compositions.

A related approach has been presented for Model Based Software Components (MBSC) in [3], introducing a component runtime environment for interpreting a domain specific model specified by this MBSC.

2 Generic runtime environment

In the original concept one MBSC is divided into two parts. The functional part consists of models of all three domain aspects as depicted in one component of Fig. 1. Different views on the model are used for the specification of data or the domain specific behavior. If required, another structural view can be used for the specification of a user interface.

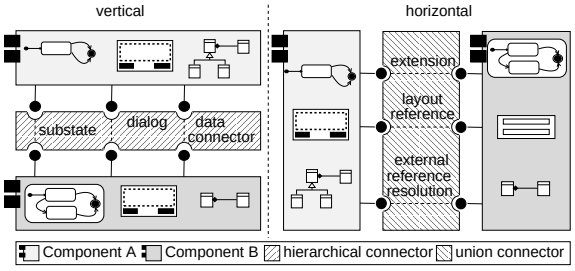


Figure 1. Functional components and their compositions

Besides the definition of view specific elements also the definition of corresponding connectors is provided in a MBSC metamodel. These connectors have to connect all three functional model views as well. The functional part is distributed inside of a technical component, representing the other part of a MBSC. While composition of MBSCs is done at the level of functional parts, the interfaces provided by the technical component are only used by a MBSC runtime node for configuration purposes. The inner architecture of a runtime node relies on several instances of the EntityContainer (EC) [2], which is a model driven object oriented data cache enabling runtime usage of the four level metamodel hierarchy. This metamodel hierarchy is defined by the Object Management Group(OMG).

In this paper the next generation of the MBSC runtime node architecture is presented, which is driven by the modularization of the MBSC metamodel and the controller layer presented in [3]. The components used in this generic runtime node architecture are depicted in Fig. 2. The OMG metamodel hierarchy is reflected by the horizontal layers, each containing an EC instance managing the elements of

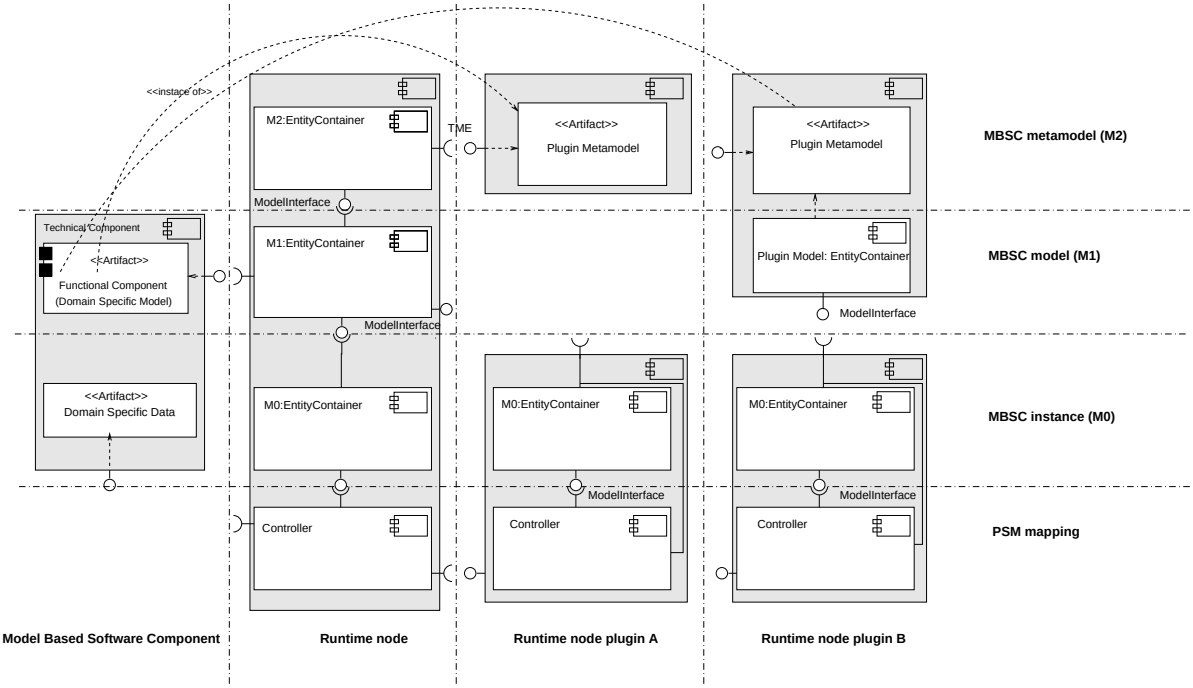


Figure 2. Architecture of a generic runtime node for functional components

this layer and providing an interface to work with the corresponding model elements in the next higher layer. The levels of distribution are illustrated by the vertical sections.

The runtime node section is required to be initially deployed on each platform used for executing MBSCs. The MBSC core metamodel defining MBSC management elements (e.g. version number, identifier and connectors) is contained in the EC of the runtime node situated in the layer M2 of Fig. 2.

In the setup phase of an executed MBSC the required model views of the MBSC are analyzed. Each model view is contributed by a plugin specific metamodel (situated at the M2 layer in Fig. 2), which is dynamically loaded and linked to the MBSC core metamodel. Also a component providing a platform specific controller (e.g. to map the model to a userinterface technology) is initialized for each loaded metamodel plugin (corresponding to one runtime node plugin column in Fig. 2).

In the next step of the setup process the connected MBSCs of the executed MBSC are analyzed and corresponding runtime node plugins are loaded (for a horizontal composition) or a new runtime node instance is started (for a vertical composition). Having finished the setup the domain specific data are loaded in the corresponding M0 EC instances of the plugins and the MBSC is executed.

3 Evaluation

The presented approach is evaluated in two use cases. The first use case is realized in the construction of a client for a warehouse management system (WMS) executed on mobile handhelds. The second use case is realized in an RFID middleware by enabling the execution of domain specific models inside an RFID reader (e.g. support shared usage of the reader by different administrative realms).

References

[1] Peter Herzum and Oliver Sims. *Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[2] G. Schmoelzer, S. Mitterdorfer, C. Kreiner, J. Faschingbauer, Z. Kovács, E. Teiniker, and R. Weiss. The entity container - an object-oriented and model-driven persistency cache. In *HICSS*. IEEE Computer Society, 2005.

[3] Michael Thonhauser, Christian Kreiner, and Martin Schmid. Interpreting Model-Based Components for Information Systems. In *Proceedings of the 16th Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, San Francisco, CA, USA, April 2009. IEEE.

Data Model Driven Enterprise Service Bus Interceptors

Michael Thonhauser[1,3], Christian Kreiner[1], Egon Teiniker[2], Gernot Schmoelzer[3]

[1]Institute for Technical Informatics
Graz University of Technology, Austria
Inffeldgasse 16, A-8010 Graz
michael.thonhauser@TUGraz.at

[2]FH JOANNEUM
University of Applied Sciences
A-8605 Kapfenberg, Austria

[3]Salomon Automation GmbH
Friesachstrasse 15
A-8114 Friesach bei Graz, Austria

Abstract

Integration of distributed software systems is an important issue in enterprise computing. Assembling of loosely coupled services via XML based protocols is a frequently used technique today.

To overcome the struggle between safety of a strong typed interface and flexibility of generic parameters, we present a novel approach that uses model-typed interface parameters together with the idea of model compatibility verification. It respects separated ownerships of service provider and consumer interfaces, and adds a mediating connector based on platform-independent, model-based functional interface reconciliation.

Given a pair of compatible interfaces an interface connector that integrates related services can be realized automatically. The concept of rule-based compatibility verification can also increase the efficiency of service repository lookups significantly.

Keywords: Service orientation, Data modeling, Model-driven development, Component-based software engineering

1. Introduction

Component Based Software Engineering (CBSE) is a mature research topic, which is accepted as a practical approach to tackle the problem of defining reusable program units to ease the development and maintenance of large distributed software systems.

In the last years different component models have been propagated to be used in various fields of software development, such as embedded systems or enterprise systems [5]. While some component models, such as Java Beans, are restricted to a specific technology, some others are platform independent like the CORBA Component Model (CCM).

Platform independence is also a key concept of the Model Driven Development (MDD) Approach [13], which aims to make use of models at different abstraction levels. This supports the development of software, which often starts at a very abstract level and gets more detailed during the software development process.

Combining the MDD and the CBSE approach leads to promising results in the development of software products, which is a key idea of the software product line development (SPL) approach [10]. In this concept the construction of software for different projects out of a family of components is supported, which are selected and configured according to a project specific set of features.

To create a software product out of distributed components the approach of Service Oriented Architectures (SOA) [4] can be used. The main idea here is to support the construction of loosely coupled software systems, using components running on heterogenous hardware and software architecture together. Current SOA systems are implemented using webservices and XML technologies.

2. Motivation

Although there exist various definitions of a Service Oriented Architecture, they have all a SOA constructed as a heterogenous distributed system in common. A key aspect of a SOA are its services, which realize the functionality

according to a business process. As communication with services is based on their interfaces, they can be seen as an instance of a software component, which is also defined as a “unit of execution with well defined interfaces” [14].

While exact interface definitions in CBSE are a key factor for fulfilling the requirement of increased reusability of components, they are also needed in SOA to support the concept of loose coupling of services. In both approaches finding the right granularity of interfaces is a non trivial task. Especially in a SOA system, which is typically a distributed system, fine grained interfaces can require a lot of service calls, leading to poor performance. These performance issues are also resulting from the currently dominant approaches of building SOA systems using XML based protocols. Because XML provides a large amount of redundant information in its tags, using this protocols leads to a large amount of transfered data or to a higher computing time, if this data is compressed for transfer purposes. Having the right granularity of interfaces, the next problem to solve is the definition of data types used by the methods of the interfaces.

To minimize dependencies, only basic data types should be used in interfaces, which leads to long parameter lists. Using complex data types in the interfaces makes reusability and interoperability difficult, because the consumer of an interface typically needs to map the complex data structure to its own. Also current approaches are lacking the support for checking compatibility of the interface data types, to support the programmer in the mapping of the missing data types. Another problem is often the need to copy the transferred data from a program data object to a webservice defined data transfer objects and vice versa.

Another problem with statically typed interfaces is the decreased flexibility of the consumer to changes in the interface, because changed interfaces are not binary compatible any more. A naive fix would introduce new data types containing the modified attributes.This quickly leads to an explosion of data types and methods in later versions of a component. A solution for this problem would be to use dynamical interfaces; unfortunately, this is also currently not supported by common tools.

Using a data model for each interface parameter eases the problems described, because the content of the generic data types of an interface can now be checked according to their model. Moreover, the interface can now be checked for compatibility by applying compatibility rules on data models of parameters.

2.1. Interfaces and their implementation

Loose coupling, as advocated by ”service orientation”, aggravates the problem of interface inconsistency. Implementors of service providers and consumers are less often

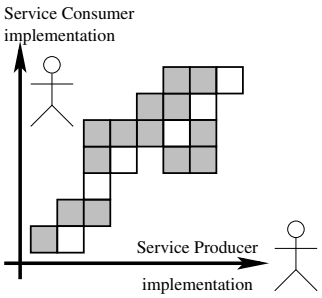


Figure 1. Independent interface implementation variation during system evolution and/or product line variabilities lead to matching, compatible, and incompatible version ranges. Only the simplest case of one interface, and one consumer and producer implementation, respectively, is illustrated.

part of the same organization compared to tighter coupled component systems. The consequence is that both implementations – owned by separate parties – evolve separately, and effort has to be put into tracking versions and their dependencies (Fig. 1). In practice, measures have to be taken, to control and engineer version ranges for interface compatibility. Yet, in real-life systems with lots of versioned interfaces, situations can occur, where no valid assembly can be found for a certain interface implementation version.

Even worse, variations of interfaces might not only occur in time during system evolution, but also due to – possibly coexisting – product variants in software product line environments [10].

Looking deeper into these symptoms, two different abstractions of an interface specification can be seen:

Functional interfaces capture the semantic aspect of an interface specification. It is used to interpret a service/component request in terms of the realization domain to exhibit functionality as required. Variations are practically unavoidable due to changes in and evolution of the component’s logic – necessitated by changes in (mostly functional) requirements. In any case, the functional interface has to be mapped in a defined way to a technical interface representation.

Technical interface. To interoperate, the technology-specific interface implementations have to fit together. The interface specification can be syntactically checked and can be used as input for code generators.

The actual design has to find a compromise between explicit parameter specification enabling com-

piler checks, and generic interfaces using opaque parameters. Explicit parameters enable rigid type-checking, mostly at compile-time; however, every change implies a new and different version of that interface. In larger systems, this can quickly become intractable. Interfaces with generic parameters ensure the technical interoperability of all components utilizing this strategy – much like the compatibility of LEGO blocks. It yields more stable interfaces, thus less variations and less trouble with incompatible versions. As a tradeoff, there is less type-safety.

This paper introduces a compatibility framework for webservices based on their parameter's data models, aiming at extended compatibility ranges. It respects separated roles and ownerships of service provider and consumer interface implementations, as well as a development role for a mediating connector based on platform-independent, model-based functional interface reconciliation. Section 3 provides an overview of related work in SOA and CBSE approaches and Section 4 introduces the concept of Model-typed interfaces.

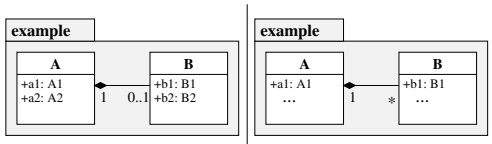
3. Related work

3.1. Service Oriented Architecture

According to Josuttis a SOA is "an architectural paradigm for dealing with business processes distributed over a large landscape of existing and new heterogenous systems that are under the control of different owners." [4]. The key technical aspects of a SOA are services, interfaces and loose coupling. To support loose coupling at the data model level only simple common types should be used. These data types should be provided by the services, which define the interface for multiple messages exchanged by the provider and the consumer in a business process. Complex data types used in the implementation of a service should be mapped to the simple data types provided by the service interfaces. All services of a SOA are connected to the Enterprise Service Bus (ESB), which is responsible for delivering the service calls. The connection of these services can be performed at different stages of coupling, ranging from fixed coupled services with fixed endpoints to loosely coupled services with mediators or interceptors. Using interceptors provides a high level of flexibility, because each service is tightly coupled to its belonging interceptor, but the interceptors are loosely coupled.

3.2. Data modeling

Model Driven Development (MDD) [13] is an approach to implement a software system by describing it in a Platform Independent Model (PIM). A PIM defines associations between data and behavior of the software and it is used as input for generators producing a platform specific model (PSM). To support MDD the Object Management Group (OMG) has released the Model-Driven Architecture (MDA) containing standards allowing specification and transformation of models.



(a) Model M_A (b) Compatible model M_B

Figure 2. Example of two different models where M_B is not a submodel of M_A, but still compatible

Models of software design are often specified using Unified Modeling Language (UML) [8], another standard of the OMG. UML models are based on a metamodel and are situated in the user model layer of the four-level metamodel hierarchy [7]. UML describes several diagrams, which can be used to model different aspects of a software. Structural and behavioral diagrams are differentiated. One example of a structural diagram is the class diagram. It is used to model the structure of classes, such as attribute and methods, as well as the association between the different classes in the model.

For data modeling purposes the metamodel of a class diagram can be extended focusing only on class attributes and associations [1].

Agile Data Modeling relies on iterative construction of data models, where each data model satisfies the requirements needed in the current iteration. It is best suitable for applications, that rely on relational databases for persistent data storage. Agile Model Driven Development (AMDD) also uses an iterative approach, instead of extensive models being generated in the normal MDD process.

3.3. Model-Typed Interfaces

The concept of Model-typed interfaces has been presented in [12, 11]. In this concept each parameter of an interface is represented by a data model, which allows to handle complex object graphs as a single parameter.

Variability analysis of compatible data models allows to define a set of rules for object-oriented model constructs, based on the metamodel presented in [11] that must be ensured for compatibility. The terms M_B as compatible model to M_A are used to describe these rules:

R_1: Classes are optional in compatible models, but each class in M_B must be available in M_A, and the package structure (namespace) of classes must be equivalent. Based on our metamodel this means for each `Entity` class in M_B an equivalent class in M_A must exist.

R_2: Attributes may vary between compatible models, but each attribute that occurs in M_B must also occur in M_A. Types of both attributes must be equivalent. However, type constraints, such as for instance maximum length of strings, can be different. In such a case constraints of the compatible model M_B need to be stronger. Regarding the metamodel a corresponding `Attribute` must exist in M_A for each `Attribute` of an `Entity` class in M_B.

R_3: Associations between classes are also optional for compatible models. As for attributes, all associations contained in M_B must be also contained in M_A. Associated classes and association types (association, composition, etc.) need to be equivalent, whereas cardinalities may differ. According to the metamodel this rule requires an `Association` in M_A for each `Association` in M_B, where types of `AssociationEnds` must be equivalent and cardinalities can vary according to rule R_{iv}.

R_4: Cardinalities of corresponding `Associations`, defined as multiplicity attribute of class `AssocationEnd` in the metamodel, can vary as long as the cardinality in M_B is more general than in M_A. For instance, cardinality of $*$ is compatible to 1 because any operation applied on a list of entities can also be applied on a single entity.

R_5: Subclasses modeled in M_B need to occur also in M_A. However, M_A may contain subclasses that are not included in M_B. In the metamodel, class inheritance is defined by `Generalization` objects between `Entity` elements.

R_6: A one-to-many composition in M_A is compatible to an association in M_B. The association's cardinality at the end of the diamond in M_A may be either 1 or 0..1. In the metamodel this means that the association kind attribute of class `AssociationEnd` may change between compatible models.

3.4. Component Connectors

The idea to use software adapters for connecting components with not identical interfaces has been discussed in various related work. A fairly comprehensive taxonomy of software connectors can be found in [6]. Most applicable in our case are the connector types "data access" and "adapter", out of eight type categories presented.

We would like to mention a paper by Yellin and Strom that describes a formal protocol specification and its usage for generating well formed component adapters in [17].

Garlan et al. [3] identify a variety of problems arising when composing large software systems and state guidelines to avoid these. Our experience described in the motivation section turned out to be largely congruent to their analysis.

For component connectors in particular connector approaches for behavioral descriptions (SIDL, [16]) can be found as well as for formal syntactic descriptions (e.g. [9, 17]), and autonomous composition of components [2].

4. Model Based Interceptors

As demonstrated in the metamodel for Model-defined interfaces [11], the data model of a parameter is defined as a partial model of the implementation specific data model of the component. Using this approach and the rules specified for checking data model compatibility, it is not required, that the data models of the using and providing component are identical; but direction of the parameter directly influences the data model compatibility. *INOUT* parameters are requiring an identical parameter data model, while *IN* or *OUT* parameters allow differing parameters, as long as compatibility rules are fulfilled.

4.1. Interface Ownership

In a software development process component interfaces are often defined by the developer or are based on a specification defined by a central organisation. An example of such an organisation is the OSGI Alliance, which defines the OSGI framework specification. An implementation of this specification is provided by the Eclipse Equinox project, which serves as a basis for Eclipse based products. Using the extension points and interfaces defined by the Equinox and Eclipse framework it is possible to integrate self developed plugins in Eclipse applications. To provide a stable interface, the publicly available interfaces often do not provided the full functionality of the framework, making customized reuse of functionality difficult.

A software developer using an interface is engaged to retrieve the interface, and is responsible on its own to check for interface updates, which might break the functionality of the interface usage. As outlined in Sect.2, handling interface relationship can become difficult, due to evolution and functional variation. Especially in the case of loose coupling of components, which is a central aspect of a SOA approach, this problem is dominant.

Also in the case of model-typed interfaces, the problem still exists. Because of the usage of a model-typed parameter, interfaces are still syntactically equivalent, even when

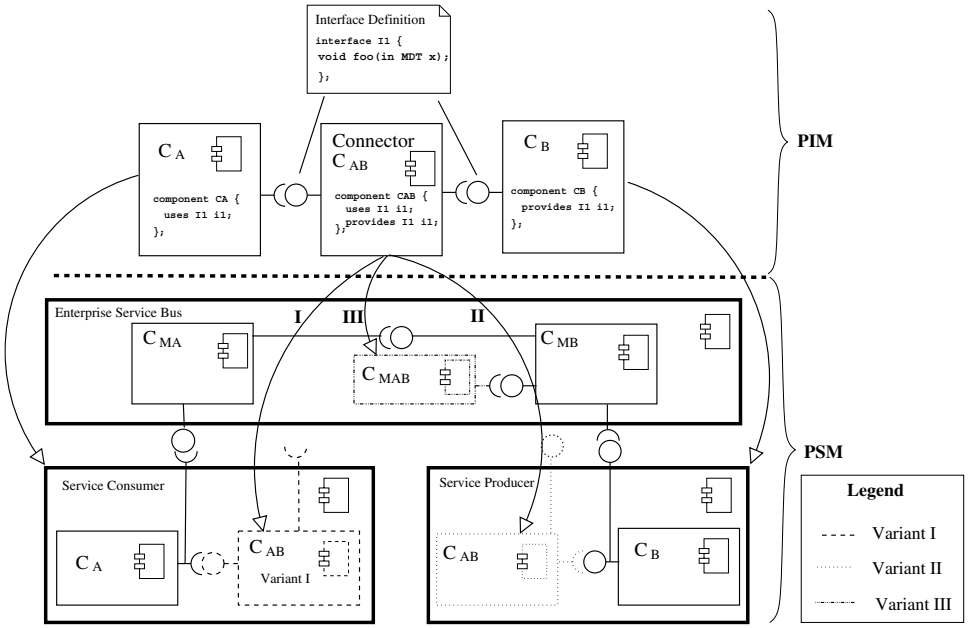


Figure 3. Variants of connector realization

the data model changes. Using the rules of model compatibility, some changing situations (such as a no longer existing required attribute) can be handled, but changes requiring additional data to be present at runtime lead to a runtime error.

For solving this problem, we apply both parameter models to a mediator component at the PIM level as demonstrated in Fig. 3. At the PSM level this mediator component is realized using an Enterprise Service Bus (ESB), with each component being connected to an ESB interceptor represented by the mediating components C_{MA} and C_{MB}. These mediating components are responsible for mapping the interface methods to messages consistent with the ESB internal protocol and they can be used for dealing with incompatible data models.

4.2. Handling data model variants

To overcome parameter data model incompatibilities, the connector C_{AB} is introduced at the PIM layer. Its PSM mapping can alternatively reside in one of the Service Producer, Service Consumer, or Enterprise Service Bus interceptor boxes. Note that this also fixes the ownership of this connector (Fig. 3).

As outlined above incompatibility of data models is often a result of adding an additionally required *IN* parameter or removing a required *OUT* parameter. For dealing with incompatibility the following variants can be distinguished:

Variant I: If a user of an interface is aware of the model incompatibility, he can implement a mediator component on its own, which performs operations to retrieve the additionally required data. This case is depicted by the nested component C_{AB}, which is assembled in the Service consumer component in Fig. 3.

Variant II: In case the provider is aware of a data model change, which might break the compatibility of the interface parameters, he can provide a mediator component, which is depicted as the nested component C_{AB}.

Variant III: Handling data model incompatibilities can also be done by an implementation of the PIM connector component provided by a third party. This use case is realized in a SOA approach, where an Enterprise Service Bus (ESB) is the basic connection mechanism. To realize loose coupling in an ESB, interceptors can be used, which are associated to one connected instance of a service enabled component (demonstrated by the components C_{MA} and C_{MB}). If in such a scenario a data model element causes incompatibilities, the ESB component being aware of this problem can query an ESB internal interface registry for a method

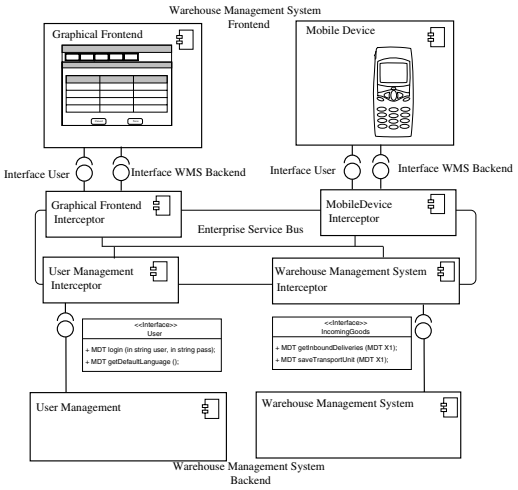


Figure 4. Example system architecture

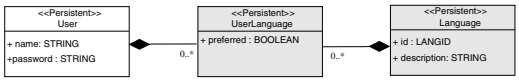


Figure 5. User data model

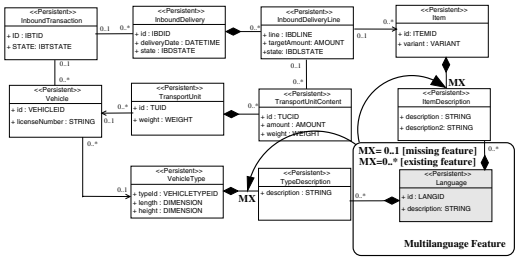


Figure 6. Warehouse Management data model

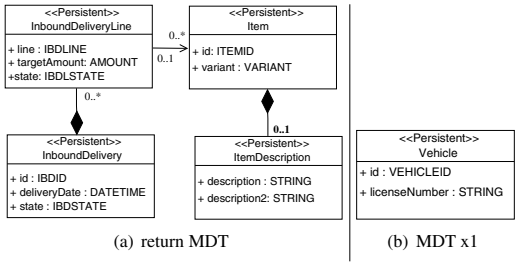


Figure 7. MDT of IncomingGoods.getInboundDelivery

to deliver the required instances of the missing model parts.

5. Example

To demonstrate our approach we evaluate a simplified warehouse management system (WMS) depicted in Fig. 4. This system consists of a WMS backend component, a user management component, a WMS graphical frontend and mobile devices handled by the warehouse workers.

Note that an implementation of these four components can be provided from different developers (e.g. different departments in a company).

The data model used by the implementation of the user management component is depicted in Fig. 5 and the WMS data model is presented in Fig. 6.

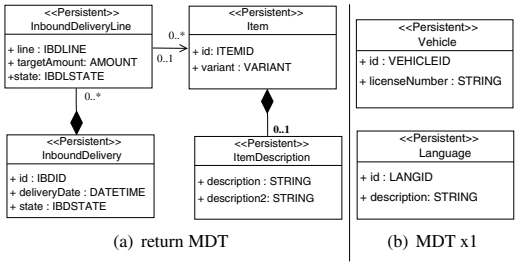
According to Fig. 6 *TransportUnit* instances are the central elements of a WMS. A number of *TransportUnitContent* instances is associated with each *TransportUnit*. Each of these *TransportUnitContent* instances again is associated with an *Item*, describing the type of the content.

Fig. 7 shows the model-defined parameter types for the method **getInboundDelivery()** in the interface **IncomingGoods** of the WMS. Note that these data types are used by the first version of the WMS, where a multi-language feature is missing in the frontend components and the WMS component. This means, that there exists no class *Language* in the corresponding data model and at most one *ItemDescription* to each retrieved *Item* (which implies MX=0..1 as depicted in Fig. 6) .

Fig. 8 shows the model-defined type (MDT) of the method **getInboundDelivery** in the interface **IncomingGoods** of the WMS with an existing multi-language feature. In this version the class *Language* and its compositions to *TypeDescription* and *ItemDescription* have been added to the data model of the WMS. Also the multiplicities at the *ItemDescription* and *TypeDescription* composition have been changed from 0..1 to 0..*, because for each language a corresponding description can be maintained for the associated *Item* and *VehicleType* instance.

The presented method is used in an incoming goods process, which is handled using the graphical WMS frontend and the mobile devices of the warehouse workers. Fig. 9 displays the sequence of tasks performed in this process.

In the beginning, the user has to log in to the WMS. In the first version of the WMS a realization of a multi-language feature is missing in the corresponding data model (i.e. the method getInboundDelivery uses the MDT pre-



(a) return MDT (b) MDT x1

Figure 8. MDT of IncomingGoods.getInboundDelivery with multi-language

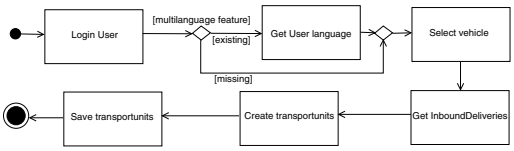


Figure 9. IncomingGoods process

sented in Fig. 7), therefore the second task is omitted. Having selected a vehicle, the corresponding *InboundDelivery* is retrieved using the method **getInboundDelivery** of the interface **IncomingGoods**. According to the MDT depicted on the left side of Fig. 7 also the corresponding *InboundDeliveryLine* instances and the associated *Item* and *ItemDescription* is returned by this method. This information is presented to the user and is used for creating the corresponding *TransportUnit* and associated *TransportUnitContent* instances for the delivered items. The created *TransportUnit* is saved by calling the method **saveTransportUnit** of the **IncomingGoods** interface.

If the data model of the WMS is extended to support a multi-language feature, it can not be guaranteed that both frontend components use the latest version of the **IncomingGoods** interface. As depicted in (Fig. 8) the method **getInboundDelivery** of the WMS backend system additionally requires a *Language* instance to be specified as an incoming parameter. Without the usage of the proposed ESB interceptors, this changed MDT would break the functionality (i.e. a runtime error would by thrown by the data model compatibility checker), if being called by a component based on the WMS data model with the missing language feature.

Using the ESB interceptors depicted in Fig. 4 the calling interceptor gets aware of the problem by checking provided and required data models according to the compatibility rules described in 3.3. As a result of this compatibility check a temporary data model is available containing all missing model elements needed for satisfying the compatibility rules. Having this information, the interceptor can look for a *compatibility data model*, which contains the information specifying a method being used for retrieving the missing data via the ESB. This *compatibility data model* can be searched for in differend places following the variants described in 4.2: the local interceptor, the opposite interceptor and finally the entire ESB. Note that this compatibility data model needs to exactly match the temporary data model; this constraint has to be defined in order to try to avoid situations, where several compatibility data models would match.

In this example the corresponding interceptor for the frontend component has a compatibility data model containing the *Language* class with an association to the method **getDefaultLanguage** of the interface **User**, which provides a compatible outgoing MDT containing a default *Language* instance for the system.

Looking at the implementation of the ESB interceptors, a big potential for automatic generation of these artifacts seems possible. These assumptions are based on the facts that method calls are mapped to ESB internal messages, which can be generated according to the data model provided, and the lookup of compatibility data models also relies on standardized methods. These *compatibility data models* can be specified using the Transient Model Extension (TME) mechanism [15]. A TME allows the specification of additional model elements, such as classes, attributes and associations. These elements are added to the extended model at runtime, and can be configured with action handlers being activated on accessing such an extended element.

If no matching method providing the required MDT is found, an interceptor for this method can be generated and connected to the original incomplete interceptor by providing a corresponding compatibility data model. Note that this implementation is afterwards available for all interceptors in the ESB sharing the same compatibility problem.

6. Conclusion

In this paper we have presented an approach for dealing with different variants of an interface at the component level. Variability of an interface is a result of its implementation variation during system evolution and/or product line variabilities. To handle these variabilities, using manually implemented software adapters is common practice. Because this process can only start when an interface mismatch is detected, it can break the functionality of the running system.

To deal with different data types used in an interface,

the concept of model-typed interfaces has been developed, which makes use of a data model to define the type of each model-typed parameter in an interface method. Using this approach allows to apply compatibility rules at the data model level of each interface parameter. Checking these rules allows to detect incompatible interfaces based on their parameters. Because a modification of the data model is not changing the syntax of a component interface, two syntactically equivalent interfaces can be connected, which rely on different implementation data models.

If the data model compatibility rules fail, manual interaction is still required by the model-typed interface approach. To support automatic lookup of components providing the missing data, a component connector has been introduced at the PIM level. This connector can be realized by an Enterprise Service Bus (ESB), which makes use of interceptors, providing protocol mapping and data model checking capabilities. If the data model compatibility rules fail during serialization or deserialization, these interceptors can make use of the functionality of an ESB by calling the appropriate method, which can be provided by another interceptor.

Future work should include modeling and development of compatibility rules for behavioral aspects of component interfaces to come closer to a comprehensive model of component interfaces. On this way – as a practical benefit – it should become more and more possible to automatically look up the best matching service in order to resolve model-typed interface incompatibilities.

References

[1] Scott W. Ambler. *Agile Database Techniques*. Wiley Publishing, Inc., 2003.

[2] Antonio Bucchiarone, Andrea Polini, Patrizio Pelliccione, and Massimo Tivoli. Towards an architectural approach for the dynamic and automatic composition of software components. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 12–21, New York, NY, USA, 2006. ACM.

[3] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185, 1995.

[4] Nicolai M. Josuttis. *SOA in Practice - The Art of Distributed System Design*. O'Reilly, first edition, August 2007.

[5] Kung-Kiu Lau and Zheng Wang. A taxonomy of software component models. *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 88–95, 30 Aug.-3 Sept. 2005.

[6] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE*, pages 178–187, 2000.

[7] OMG. UML Infrastructure, Version 2.0 . Technical Report 2002-09-01, Object Management Group, 2002.

[8] OMG. UML Superstructure, Version 2.0. Technical Report 2002-09-02, Object Management Group, 2002.

[9] Niloofar Razavi and Marjan Sirjani. Using reo for formal specification and verification of system designs. In *MEMOCODE 2006, Napa, CA*, pages 113–122, 2006.

[10] Gernot Schmoelzer, Christian Kreiner, and Michael Thonhauser. Platform design for software product lines of data-intensive systems. In *EUROMICRO-SEAA*, pages 109–120. IEEE Computer Society, 2007.

[11] Gernot Schmoelzer, Egon Teiniker, and Christan Kreiner. Model-typed component interfaces. *Journal of System Architecture*, 54(6):551–561, June 2008.

[12] Gernot Schmoelzer, Egon Teiniker, Christian Kreiner, and Michael Thonhauser. Model-typed component interfaces. In *EUROMICRO-SEAA*, 2006.

[13] Thomas Stahl and Markus Voelter. *Model-Driven Software Development*. Wiley, 2006.

[14] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Component Software Series. Addison Wesley, 2002.

[15] Michael Thonhauser, Gernot Schmoelzer, and Christian Kreiner. Model-based data processing with transient model extensions. In *ECBS*, pages 299–306, 2007.

[16] Lei Wang and Padmanabhan Krishnan. A framework for checking behavioral compatibility for component selection. In *ASWEC 2006*, pages 49–60, 2006.

[17] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.

A Model-Based Architecture supporting Virtual Organizations in Pervasive Systems

Michael Thonhauser, Christian Kreiner, Andreas Leitner
Institute for Technical Informatics
Graz University of Technology
Graz, Austria
michael.thonhauser@TUGraz.at

Abstract

Modern distributed computer systems, with mobile and embedded devices as first class citizens, are formed from heterogeneous platforms. Owing to their distributed nature, dynamic reconfiguration and adaptation are reflecting different ownerships and administration domains of devices and applications.

A portable, plug-in extensible runtime architecture is presented that explicitly honors ownership and realm of control of hardware devices, resources and application components. Based on such an architecture, their owners, while pursuing their very own business models, can cooperatively form Virtual Organizations (VO) to run applications defined by model-based software components (MBSC), consisting of a set of high-level models that are directly interpreted by this architecture's runtime nodes.

1. Introduction

Today we are confronted with an increasing number of distributed systems made up of mobile and embedded devices integrated in their surroundings [1]. Performance and storage capacities of these devices is steadily increasing. This enables them more and more to take part in distributed systems as first class members. Still, applications for pervasive systems have to deal with platform heterogeneity and resource constraints of such devices.

As first class members, devices are expected to share their resources (e.g. CPU, memory, I/O resources) with other distributed system members. Support for authentication, authorization and billing is required for this feature, which is provided by the concept of Virtual Organizations (VO) developed in the context of Grid Computing.

Current architectures supporting mobile and embedded devices (e.g. Cloud Computing) are build upon Internet technology and the execution of application on centralized servers. Following this client-server paradigm, mobile and embedded devices are often used as simple sensors or visualization devices only. In contrast, Foster et.al. [2] have underlined the importance of Client Computing to overcome restraints regarding security and performance in their discussion of Cloud Computing.

2. Related Work

Grids are a form of distributed systems (defined as a collection of independent computers that appears to its users as a single coherent system [1]) that coordinate distributed resources using standard protocols and interfaces to deliver nontrivial quality of service [3]. Virtual Organizations (VO) in Grids should support a dynamic group of users with a common goal – coming together for a specific and short-lived collaborative venture. Unfortunately, this has never been realized owing to the complexity of deploying and authorizing such a dynamic structure [4]. However, Grid use cases and the understanding of VOs [5] are changing by emerging Grid approaches [6]. With Cloud Computing, another approach for construction of distributed systems has been proposed recently. It enables dynamic resource usage through virtualization at different levels of abstraction: Services, Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service [7]. The main drawbacks of this approach are the central data storage and the required network connections.

Software models providing the abstraction required for distributed heterogeneous systems development are standardized in the four-level meta-model hierarchy of the Object Management Group (OMG). Each level (named M3 to M0) defines the valid model elements of the next lower layer. While traditional modeling languages like the Unified Modeling Language are defined following this paradigm, other approaches such as Domain Driven Design [8] make use of custom models at each layer to capture domain specific information and to foster the usage of one model throughout the entire system development process. The OMG hierarchy is also applied by Model Driven Software Development (MDSD) [9] approaches, mapping a platform independent model (PIM) to a platform specific model (PSM) by the help of generators at system development time, or with model interpretation at runtime.

Recent approaches for using models at runtime are based on the understanding, that "a model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from the problem space perspective." [10].

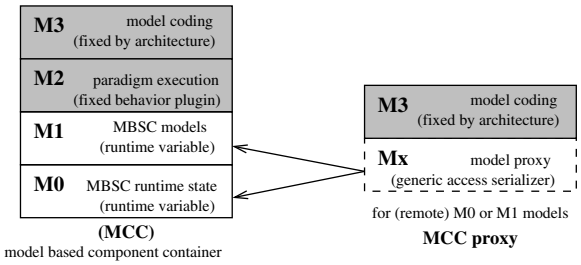


Figure 1. Model based Component Container (MCC) and MCC proxy

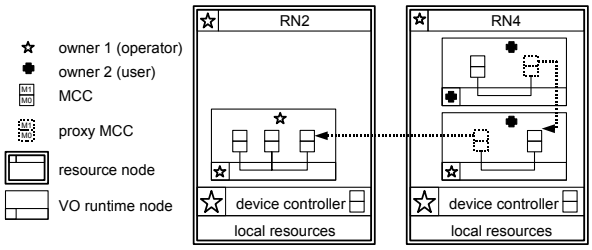


Figure 2. Nodes and owners in a model based VO

3. Model-based software components

Creating data entities based on the interpretation of data models at runtime is a key aspect of the EntityContainer (EC) [11] approach. To support other models (e.g. Statecharts to specify behavior), the EC concept has been extended to a Model-Based Component Container (MCC) [12]. Fig. 1 shows the stacked data stores of an MCC, where each one manages one layer of the OMG meta-model hierarchy. The topmost M3 model is constant, defined as part or the architecture, and is used for generic encoding of all kinds of models in this architecture. The also static M2 model defines the view to be used by the entities. Their correctness is ensured by M2 specific validators used in the MCC. Optionally, the MCC model content at M0 and M1 can be interpreted by a controller implementing behavior for the domain described by the governing M2 model.

MCCs support transactional updates with a 2-phase commit cycle, thus avoiding (temporarily) inconsistent system states during updates in several - possibly remote - MCCs due to a state changing transaction.

Remote MCCs are locally represented by MCC proxies (Fig. 1). These provide local interfaces to MCCs executing in another VON (see below) implementing an MBSC's receptacle. An MCC proxy gets connected to a target MCC by configuring it with the the latter's uniform resource name. They make use of the constant M3 model for generic validation and encoding purposes.

Model-Based Software Components (MBSC) [13] consist of a set of models. Each of these models defines a specific domain aspect of the MBSC: its class model specifies data structures and/or user interfaces, while a state model defines its behavior, etc. Several MCCs cooperate for executing the set of models provided by a MBSC at runtime. All required MCC types, together with their M2 specific model interpreters, are plugged into the VON executing the given MBSC on demand.

The public ports of MBSCs are provided as model defined interfaces [14], also utilizing models for the definition of the view through provided and required interfaces. In this concept both roles – with potentially different owners behind – have their own interface perception, i.e. are the owner of their interface models. In this way, MBSC internal contents are self-contained in term of ownership and integrity. Because of the weaker type safety induced by the model compatibility rules for connecting model defined interfaces, asynchronous MBSC evolution in separated owner realms is supported, at the cost of necessary checks at runtime.

4. Model-based virtual organizations

A basic rule of the presented approach is, that each part of this architecture viz. application components, runtime containers, resource nodes and resources has a clearly defined owner that does not change at runtime. An owner is defined as an identification of a managing authority (possibly a larger organization). This owner has the means and rights to manage the life cycle, development and evolution of an application as well as access rights to resources and the applications business model. These model-based applications are defined using several MBSCs. Their deployment manifests a VO made up by ≥ 1 owners bound by a VO (collaboration) contract.

The runtime architecture is depicted in Fig. 2 and is made up of the following parts.

(Local) resources are directly connected I/O resources, which are not part of this architecture (like UI-hardware, sensors, auto-id devices (RFID readers)), and local computing resources (e.g. CPU cycles, memory). All resources are abstracted to resource allocation rights (e.g. create/destroy actions), that can be granted to a certain MBSC and its owner in turn.

Resource Nodes (RN) are executed on a local machine and are containing a device node controller attaching all local resources to the architecture. This information is provided by a MCC being part of the device node controller, which is configured with a metamodel as depicted in Fig. 3.

VO runtime nodes (VON) can host MCCs to accommodate and execute MBSC's of one tenant owner only. Note that the owner of the VON itself – who has created it – can be different from the content owner. In this

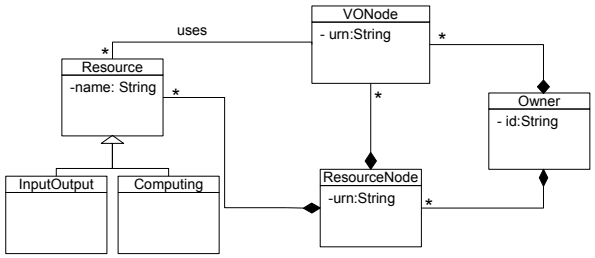


Figure 3. Model used by Device controller MCC

way, the tenant's realm of control is isolated from other owners in the system. A VON features a controller to create MCCs and MCC proxies required by the residing MBSCs. The information of this controller is also managed by a specific MCC.

Node identification is accomplished through a uniform resource name (URN). Both URNs and also URLs are Uniform Resource Identifier (URI), whereas the former is used for identification and the latter for locating or finding resources. The essential difference between both can be described as "what" vs. "where". As defined in [15] all URNs have the following syntax (phrases enclosed in quotes are required):
<URN>::= "urn:" <NID>":" <NSS>
where <NID> is the namespace identifier, and <NSS> is the namespace specific string. A sample resource name for a runtime data container could be: "urn:vo:customer1:runtimeNode2:dataContainer1".

At runtime, the distributed system is formed by several RNs and their executing VONs containing the model based applications. Several use cases with respect to the adaptation of the distributed system at runtime were identified and are supported by the architecture. Besides others, there are:

VON creation by RN owner: new/initial VONs are typically created by the operator.

Value-added resellers (VAR): a customer can act as a VAR, if he is possessing a VON (created by the operator) and has the rights to create VONs. Given these rights, he can create new VONs for his customers in turn within one of his application MBSCs running in the first VON.

MBSC application execution: having started the RN and corresponding VONs, and having deployed the MBSCs, their execution is started. During execution, communication between different MCCs is possible using MCC specific queries contained in the interpreted models.

VON migration might become necessary during adaptations of the distributed system at runtime. A new VON gets created for the same tenant on another RN, and the contents of all MCCs are transferred. The new MCCs are registered for name resolution (in the RN!) within the tenant's name space. Having finished, all facets in the migrated VON interfaces will resolve to the new RN location, as well as all component receptacles using an MBSC within migrated VON.

(Partial) shutdown. The steps required to shutdown a VON are consisting of ECQL statements targeting the device controller and the specific VON. Note that a VON used by a tenant can be deleted by the owner, if the tenant does not satisfy the terms of conditions anymore. If the shutdown of a RN is required, each VON is informed to start the migration to another RN belonging to the corresponding VO.

Table 1. Statemachine models

States	Transitions	Actions	Conditions	Encoded size (bytes)
5	7	5	4	4605
2	2	1	0	881

5. Evaluation

The presented architecture has been implemented in a scenario targeting the shared use of an RFID reader. The implemented prototype has been based on the .NET Microframework, leveraging JavaScript Object Notation (JSON) for the platform independent encoding of the entities, the corresponding models and the communication protocol between several MCCs.

The RNs and VONs involved in this scenario are depicted in Fig. 2: RN4 is executed directly on the RFID reader device containing a VON responsible for detecting and preprocessing tag reads. The second VON on RN4 is created for filtering purposes of the recognized tags. If a filtered tag is detected, another VON running on another node (RN2) is informed, which signals the detected tag to the user of the hardware running this VON by initializing a beep of the system speaker.

This scenario can be applied in the business domain of logistics by letting a logistics provider (the owner of RN4) act as a VAR. As described in the use case in Sect. 4 he can allow his customer to load a filtering application into the other VON on RN4 enabling the realization of a selective track and trace scenario. In case the customer requires another device to inform him about a filtered tag read, he could simply migrate the VON running on RN2 to an RN providing the same resource statements as required by the MBSC, which defines the tag notification functionality.

Tab. 1 characterizes the size of statemachines used in the the prototype. Tab. 2 and Tab. 3 contain metrics about the models used in the class MCCs, as well as the controller MCCs of VONs and RNs, respectively. Note that these files define the application and deployment of this model-based application; accordingly they need to be transferred to their runtime environments.

Table 2. Data models (M1)

ID	Classes	Attributes	Associations	Generalizations	Encoded size (bytes)
VON	6	2	0	5	1049
RN	4	3	2	2	917
Data0	2	0	1	0	315
Data1	1	1	0	0	128
Data2	14	19	0	6	2102

Table 3. Data instances (M0)

ID	Objects	Attributes	Encoded size (bytes)
RN2	2	2	256
RN4	3	3	363
VON	8	3	560
Data0	1	0	68
Data1	1	1	133
Data2	11	19	1110

The prototypical implementation of the runtime environment requires about 215 kB of storage on the device for the implementation of the RN, VON and MCCs, while the implementation of one resource access method (which has been deployed in a separate assembly) required between 4 kB and 7.5 kB. Some performance issues were noted during the tests of the prototype, which were dependent on the hardware and the .NET MF version used.

6. Conclusion

In this paper a model-based runtime architecture for distributed pervasive systems has been presented, supporting the dynamics of virtual organizations (VO) by the separation of resource nodes (RN) and VO runtime nodes (VON). Local resources are controlled by the RNs, and made available to their VONs. VONs are used as execution platforms of model-based software components (MBSC) leveraging a set of models for platform independent specification of domain specific functionality. These models are loaded into several Model based component containers (MCC) dynamically plugged into a VON at runtime.

A prototypical implementation has been realized based on the .NET Microframework and utilizing JSON for data encoding. The prototype has been evaluated in a logistics scenario, and has been used to demonstrate various use cases relevant for this architecture (e.g. a VON owned by a VAR, migration of different VONs).

In the future we are considering the support of other runtime platforms and we plan to investigate required steps for increasing the security of the runtime platform. Furthermore the prototypical results should be evaluated in bigger scenario, such as the support of shared infrastructure in home automation systems.

Acknowledgment: This work was funded by RF-iT Solutions GmbH, Graz, Austria and FFG.

References

[1] A. S. Tanenbaum and M. van Steen, *Distributed Systems. Principles and Paradigms*, 2nd ed. Prentice Hall International, October 2006.

[2] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud Computing and Grid Computing 360-Degree Compared," in *Grid Computing Environments Workshop, 2008. GCE '08*, November 2008, pp. 1–10.

[3] C. K. Ian Foster, *The Grid. Blueprint for a New Computing Infrastructure.: Blueprint for a New Computing Infrastructure (Elsevier Series in Grid Computing)*, 2nd ed. Morgan Kaufmann, December 2003.

[4] I. Bird, B. Jones, and K. F. Kee, "The organization and management of grid infrastructures," *Computer*, vol. 42, no. 1, pp. 36–46, 2009.

[5] M. Waldburger and B. Stiller, "Toward the Mobile Grid:Service Provisioning in a Mobile Dynamic Virtual Organization," in *Computer Systems and Applications, 2006. IEEE International Conference on.*, April 2006, pp. 579–583.

[6] H. Kurdi, M. Li, and H. A. Raweshidy, "A classification of emerging and traditional grid systems," *IEEE Distributed Systems Online*, vol. 9, no. 3, p. 1, 2008.

[7] N. Leavitt, "Is cloud computing really ready for prime time?" *Computer*, vol. 42, no. 1, pp. 15–20, January 2009.

[8] E. J. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 1st ed. Addison-Wesley Professional, September 2003.

[9] M. Voelter and T. Stahl, *Model-Driven Software Development*, 1st ed. Wiley & Sons, May 2006.

[10] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *Computer*, vol. 42, no. 10, pp. 22–27, October 2009.

[11] G. Schmoelzer, S. Mitterdorfer, C. Kreiner, J. Faschingbauer, Z. Kovács, E. Teiniker, and R. Weiss, "The entity container - an object-oriented and model-driven persistency cache," in *HICSS-38*. IEEE Computer Society, 2005.

[12] M. Thonhauser and C. Kreiner, "Towards a generic model interpretation runtime architecture," in *Work in Progress Session*. Euromicro, August 2009.

[13] M. Thonhauser, C. Kreiner, and M. Schmid, "Interpreting model-based components for information systems," in *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, 2009, pp. 254–261.

[14] G. Schmoelzer, E. Teiniker, and C. Kreiner, "Model-typed component interfaces," *J. Syst. Archit.*, vol. 54, no. 6, pp. 551–561, 2008.

[15] R. Moats, *URN Syntax*, IETF, May 1997, http://www.ietf.org/rfc/rfc2141.txt.

Appendix A

Glossary

Distributed system: "A distributed system is a collection of independent computers that appears to its users as a single coherent system." [TvS06, p.2]

Embedded system: "Embedded systems are microcontroller-based systems built into technical equipment. They're designed for a dedicated purpose and usually don't allow different applications to be loaded and new peripherals to be connected. Communication with the outside world occurs via sensors and actuators; if applicable, embedded systems provide a human interface for dedicated actions.".[ES09, p.14]

Entity: "An object fundamentally defined not by its attributes, but by a thread of continuity and identity" [Eva03, p.512]

Environment: "The environment, or context, determines the setting and circumstances of developmental, operational, political, and other influences upon that system. The environment can include other systems that interact with the system of interest, either directly via interfaces or indirectly in other ways. The environment determines the boundaries that define the scope of the system of interest relative to other systems." [IEE07, p.4]

Grid: "A distributed computing infrastructure that supports the creation and operation of virtual organizations by providing mechanisms for controlled, cross-organization resource sharing." [FK04, p.662]

Logical mobility: "A client that is logically mobile is aware of its location changes. In order to relieve the client from adapting manually to new locations, the main concern of logical mobility is automated location awareness within the publish/subscribe middleware." [MFP06, p.289]

Metamodeling: Kleppe [Kle08] provides a definition of a model, which is rested upon a combination of a type graph and a set of constraints at various types of this graph. A type graph is defined as a combination of

- a set of nodes which may include data types
- a set of edges

- a source function from edges to nodes, which gives the source node of an edge
- a target function from edges to nodes, which gives the target node of an edge
- An inheritance relationship between nodes (a reflexive partial ordering)

The concept of a labeled graph is also applied in the representation of a class diagram, which is usually made use of for the presentation of elements in the M3 and M2 layer of the OMG four-level meta-model hierarchy. Having motivated the usage of graphs for the definition of a model Kleppe also defines an instance of a model M as a labeled graph that can be typed over the type graph of M and satisfies all the constraints in M's constraint set.

Mobile environment: A mobile environment is consisting of devices and applications supporting the logical and physical mobility of their users.

model@run.time: "A model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective."[BBF09, p.23]

Physical mobility: "A client that is physically mobile disconnects for certain periods of time and has different border brokers along its itinerary through the infrastructure. The main concern of physical mobility is location transparency." [MFP06, p.289]

Platform: "A platform is the combination of a language specification, predefined types, predefined instances, and patterns, which are the additional concepts and rules needed to use the capabilities of the other three elements." [Kle08, p.69]

Software Architecture: "The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution." [IEE07]

Software Component: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third party." [Szy02, p.41]

Software-intensive system: "A software-intensive system is any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole." [IEE07, p.1]

Software product line: "Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customisations." [PBvdL05, p.14]

System: A collection of components organized to accomplish a specific function or a set of functions. [IEE07, p.3]

Value objects: "An object that describes some characteristic or attribute but carries no concept of identity." [Eva03]

Virtual Organization: "A collaboration whose participants are both geographically and organizationally distributed." [FK04, p.672]

Bibliography

[ABB+01] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jurgen Wust, and Jorg Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley Professional, 1st edition, November 2001.

[AK05] Colin Atkinson and Thomas Kühne. A generalized notion of platforms for model-driven development. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, chapter 6, pages 119–136. Springer-Verlag, Berlin/Heidelberg, 2005.

[All09] OSGI Alliance. Osgi service platform, core specification, release 4, version 4.2. Technical report, OSGI Alliance, September 2009.

[BBF09] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, October 2009.

[BCD+09] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. Gcm: a grid extension to fractal for autonomous distributed components. *Annals of Telecommunications*, 64(1):5–24, February 2009.

[Ben08] Nelly Bencomo. *Supporting the Modelling and Generation of Re ective Middleware Families and Applications using Dynamic Variability*. PhD thesis, Computing Department Lancaster University, March 2008.

[BHM09] Tomáš Bureš, Petr Hnětynka, and Michal Malohlava. Using a product line for creating component systems. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 501–508, New York, NY, USA, 2009. ACM.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48, September 2006.

[BHP+07] T. Bures, P. Hnetynka, F. Plasil, J. Klesnil, O. Kmoch, T. Kohan, and P. Kotrc. Runtime support for advanced component concepts. In *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*, pages 337–345, September 2007.

[BJK09] I. Bird, B. Jones, and K. F. Kee. The organization and management of grid infrastructures. *Computer*, 42(1):36–46, 2009.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, August 1996.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework. (Eclipse Series).* Prentice Hall International, August 2003.

[CBG+08] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, February 2008.

[CD00] John Cheesman and John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software.* Addison-Wesley Professional, October 2000.

[CGFP09] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43, October 2009.

[Cho07] Yunja Choi. Checking interaction consistency in marmot component refinements. In Jan Leeuwen, Giuseppe F. Italiano, Wiebe Hoek, Christoph Meinel, Harald Sack, and František Plášil, editors, *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *Lecture Notes in Computer Science*, chapter 72, pages 832–843. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[dCYG08] C. A. da Costa, A. C. Yamin, and C. F. R. Geyer. Toward a general software infrastructure for ubiquitous computing. *Pervasive Computing, IEEE*, 7(1):64–73, January 2008.

[DJS+09] Trip Denton, Edward Jones, Srini Srinivasan, Ken Owens, and Richard Buskens. Naomi an experimental platform for multimodeling. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, chapter 10, pages 143–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[EJ09] Christof Ebert and Capers Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, June 2009.

[ES09] Christof Ebert and Juergen Salecker. Guest editors' introduction: Embedded software technologies and trends. *Software, IEEE*, 26(3):14–18, April 2009.

[Eva03] Eric J. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Longman, Amsterdam, 1. a. edition, September 2003.

[FK04] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure (Elsevier Series in Grid Computing).* Morgan Kaufmann, second edition, 2004.

[FNY09] G. Fischer, K. Nakakoji, and Yunwen Ye. Metadesign: Guidelines for supporting domain experts in software development. *Software, IEEE*, 26(5):37–44, August 2009.

[Fre05] Andreas R. Frei. *Jadabs - An adaptive pervasive middleware architecture.* PhD thesis, Swiss Federal Institute of Technology Zurich, 2005.

[FZRL08] I. Foster, Yong Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, November 2008.

[Gro09a] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Longman, Amsterdam, 1 edition, April 2009.

[Gro09b] Object Management Group. Omg unified modeling language (omg uml),infrastructure. Technical report, Object Management Group, February 2009.

[Gro09c] Object Management Group. Omg unified modeling language (omg uml),superstructure. Technical report, Object Management Group, February 2009.

[Hes09] Anders Hessellund. *Domain-Specific Multimodeling*. PhD thesis, IT University of Copenhagen, February 2009.

[Hor99] Ian Horrocks. *Constructing the User Interface with Statecharts*. Addison Wesley, illustrated edition edition, 1999.

[HP98] David Harel and Michal Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw-Hill Companies, October 1998.

[HS00] Peter Herzum and Oliver Sims. *Business Components Factory: A Comprehensive Overview of Component-based Development for the Enterprise*. Verlag John Wiley & Sons, Inc, 1., auflage edition, January 2000.

[IC02] Magnus Larsson Ivica Crnkovic, editor. *Building Reliable Component-Based Software Systems*. Artech House Publishers, 1st edition, July 2002.

[IEE07] IEEE. Iso/iec standard for systems and software engineering - recommended practice for architectural description of software-intensive systems. Technical report, IEEE, July 2007.

[Jor09] Paul C. Jorgensen. *Modeling Software Behavior: A Craftsman's Approach*. Auerbach Publications, 1 edition, July 2009.

[KLAR08] Heba Kurdi, Maozhen Li, and Hamed Al-Raweshidy. A classification of emerging and traditional grid systems. *IEEE Distributed Systems Online*, 9(3), 2008.

[Kle08] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, December 2008.

[Kre08] Stefan Kremser. Datencache für anwendungen im mobile grid. Master's thesis, Institute for Technical Informatics,Graz University of Technology, April 2008.

[Kru95] Philippe Kruchten. Architectural blueprints—the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, November 1995.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[Kuh08] Jens Kuhner. *Expert .NET Micro Framework (Expert's Voice in .Net)*. Apress, April 2008.

[Lei10] Andreas Leitner. Model-based support of virtual organizations in an rfid middleware. Master's thesis, Institute for Technical Informatics, Graz University of Technology, May 2010.

[LT09] P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *Software, IEEE*, 26(3):19–25, April 2009.

[LW06] K. K. Lau and Z. Wang. A survey of software component models. Technical report, School of Computer Science, The University of Manchester, May 2006.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.

[MBJ+09] B. Morin, O. Barais, J. M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, October 2009.

[MFB09] Pierre-Alain Muller, Frédéric Fondement, and Benoît Baudry. Modeling modeling. In *Model Driven Engineering Languages and Systems*, chapter 2, pages 2–16. Springer Berlin / Heidelberg, 2009.

[MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer, 1 edition, July 2006.

[Mil09] Dragan Milicev. *Model-Driven Development with Executable UML (Wrox Programmer to Programmer)*. Wrox, 1 edition, July 2009.

[MJR+04] R. Moore, A.S. Jagatheesan, A. Rajasekar, M. Wan, and W. Schroeder. Data Grid Management Systems. In *Proceedings of the 21stI EEE/NASA Conference on Mass Storage Systems and Technologies (MSST)*, College Park, Maryland, USA, April 2004. EEE/NASA.

[ML05] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, October 2005.

[MMC09] Pranav Mistry, Pattie Maes, and Liyan Chang. Wuw - wear ur world: a wearable gestural interface. In *CHI EA '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 4111–4116, New York, NY, USA, 2009. ACM.

[Mul93] David Mulcahy. *Warehouse Distribution and Operations Handbook (McGraw-Hill Handbooks)*. McGraw-Hill Professional, 1 edition, September 1993.

[NAD+02] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew Black, Peter Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born. A component model for field devices. In Judith Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, chapter 14, pages 1–13. Springer Berlin Heidelberg, Berlin, Heidelberg, June 2002.

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, September 2005.

[RQZ07] Chris Rupp, Stefan Queins, and Barbara Zengler. *UML 2 glasklar*. Hanser Fachbuchverlag, third edition, August 2007.

[Sch07] Gernot Schmölzer. *A Model-based Software Product Line Architecture for Data-intensive Systems*. PhD thesis, Graz University of Technology, June 2007.

[SJ09] Suzila Sabil and Dayang N. Jawawi. Integration of pecos into marmot for embedded real time software component-based development. In *2009 Fourth International Conference on Software Engineering Advances*, pages 265–270. IEEE, September 2009.

[SKKT06] G. Schmoelzer, C. Kreiner, Z. Kovacs, and M. Thonhauser. Reflective, model-based data access with the type-safe entity container. In *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*, volume 2, pages 87–92, 2006.

[SM03] Debashis Saha and Amitava Mukherjee. Pervasive computing: A paradigm for the 21st century. *Computer*, 36(3):25–31, March 2003.

[SMK+05] Gernot Schmoelzer, Stefan Mitterdorfer, Christian Kreiner, Joerg Faschingbauer, Zsolt Kovacs, Egon Teiniker, and Reinhold Weiss. The entity container - an object-oriented and model-driven persistency cache. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, Washington, DC, USA, 2005. IEEE Computer Society.

[STK08] Gernot Schmoelzer, Egon Teiniker, and Christian Kreiner. Model-typed component interfaces. *Journal of Systems Architecture - Embedded Systems Design*, 54(6):551–561, 2008.

[SV06] Thomas Stahl and Markus Völter. *Model-driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming (2nd Edition)*. Addison-Wesley Professional, 2 edition, November 2002.

[Tei05] Egon Teiniker. *A Novel Component Platform for Logistics Software Product Lines*. PhD thesis, Graz University of Technology, 2005.

[Træ08] H. Trætteberg. Integrating dialog modeling and domain modeling - the case of diamodl and the eclipse modeling framework. *Journal of Universal Computer Science*, 14(19):3265–3278, 2008.

[TvS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, October 2006.

[VS06] Markus Völter and Thomas Stahl. *Model-Driven Software Development*. Wiley & Sons, 1., auflage edition, May 2006.

[Wan09] R. Want. When cell phones become computers. *Pervasive Computing, IEEE*, 8(2):2–5, 2009.

[Wei95] Mark Weiser. The computer for the 21st century. *Human-computer interaction: toward the year 2000*, pages 933–940, 1995.