To appear in
Proceedings of the 3rd IFIP International Conference on
e-Commerce, e-Business and e-Government (I3E 2003)
22-25 September 2003, Guarujá, Brazil.

Use of Models and Modelling Techniques for Service Development

Luís Ferreira Pires[*], Marten van Sinderen[*], Cléver Ricardo Guareis de Farias[**], João Paulo Andrade Almeida[*]

[*]University of Twente (UT)
P.O. Box 217, 7500 AE, Enschede, The Netherlands
{pires, sinderen, almeida}@cs.utwente.nl

[**]Universidade Católica de Santos (Unisantos)
Rua Dr. Carvalho de Mendonça, 144
11070-906 – Santos (SP), Brazil
cleverfarias@unisantos.br

Abstract: E-applications are increasingly being composed from individual services that can be realized with different technologies, such as, e.g., Web Services and standard component technologies. A current trend in the development of these services is to describe their technology-independent and technology-specific aspects in separate models. A prominent development that leads this trend is the Model-Driven Architecture (MDA). An important feature of the MDA approach is the explicit identification of Platform-Independent Models (PIMs) and the flexibility to implement them on different platforms via Platform-Specific Models (PSMs), possibly through (automated) model transformations. A platform can be any technology that supports the execution of these models, either directly or after translation to code in a programming language. This paper aims at identifying the benefits of the MDA approach in the development of services for e-applications. The paper presents a short introduction to MDA, in the context of service development, and an overview of the modelling capabilities of the Unified Modelling Language (UML), one of MDA's main modelling languages.

Keywords: Service-oriented development; Model Driven Architecture; Unified Modeling Language.

1 Introduction

There is a growing need to compose e-applications from individual services that can be provided by both proprietary components and third-party service providers. This need arises from requirements with respect to, e.g., shorter time-to-market, reduced development costs, and reuse of proven technological solutions. The ideal of

a service-oriented development or service-oriented architecture is also fuelled by the industrial uptake of technologies such as Web Services and standard component technologies.

A current trend in the development of services is to separate their technology-independent and technology-specific aspects, by describing them in separate models. The most prominent development in this trend is the Model-Driven Architecture (MDA) [10, 13] approach, which is being fostered by the Object Management Group (OMG). The MDA approach is not a design methodology, but rather a collection of guidelines to be applied in combination with a design methodology in order to develop distributed applications. The core of the MDA consists of a number of OMG standards, including: the Unified Modelling Language (UML) [18], the Meta Object Facility (MOF) [12], and the XML Metadata Interchange (XMI) [15].

The most important aspect of the MDA approach is the explicit identification of Platform-Independent Models (PIMs) and the flexibility to implement them on different platforms via Platform-Specific Models (PSMs). A platform can be any technology that supports the execution of these models, either directly or after translation to code. In the case of distributed applications, MDA can be applied to develop PIMs that are middleware technology-independent, and develop PSMs for specific middleware platforms like CORBA/CCM, EJB or Web Services. MDA also aims at facilitating the translation from PIMs to PSMs, by introducing profiles for defining PIMs and PSMs, and by standardising transformations between them, which can then be automated by tools. Some research is being carried out in order to define these transformations and automate them.

This paper presents a short introduction to MDA and an overview of the modelling capabilities of UML, which is one of MDA's main modelling languages. The paper is further structured as follows: section 2 introduces some basic modelling concepts and principles; section 3 introduces the MDA approach; section 4 discusses the modelling of services using UML; finally, section 5 presents some final remarks.

2 Modelling principles

A *model* is a representation of structural or behavioural aspects of a system in a language that has a well-defined syntax, semantics, and possibly rules for analysis, inference, or proof [13].

Models can be used in different ways in the course of a development project. A model used to prescribe properties of a system or system part to be built is called a *prescriptive model*. In contrast, a model used to describe an existing system or system part is called a *descriptive model*. In the case of prescriptive models, designers produce models of a system introducing information that constrain the intended characteristics of the system being specified. The information required for modelling is obtained along the development trajectory, and documented in several ways.

In order to understand any non-trivial system, one has to cope with a large amount of interrelated aspects. Attempting to capture all aspects of the design in a single model yields too complex and useless models [6]. Therefore, models should be

derived using specific sets of abstraction criteria, which allow one to focus on particular aspects of the system at a time.

2.1 Viewpoints and abstraction levels

A model is often characterized by the set of abstraction criteria used to determine what should be included in the model. *Viewpoints* and *abstraction levels* are examples of abstraction criteria.

A viewpoint defines a set of related concerns that play a distinctive role in the design of a system. A model defined from a particular viewpoint focuses on the particular concerns defined by the viewpoint. Viewpoints should be chosen with respect to requirements that are of concern to some particular group involved in the design process.

Examples of viewpoints are the five RM-ODP viewpoints [7]: enterprise, information, computational, engineering and technology. The use of different viewpoints in order to describe a system raises the issue of consistency. Descriptions of the same or related entities appear in different viewpoints. Therefore, one must assure that these multiple models are not in conflict with each other.

Abstraction is the process of suppressing irrelevant detail to establish a simplified model, or the result of that process [6]. A model M_1 is at a higher level of abstraction than a model M_2 if M_1 suppresses details of the system that are revealed by M_2. Specifically, the pair of models $\{M_1, M_2\}$ is in a refinement relationship, in which M_1 (the abstraction) is more abstract than M_2 (the realization).

Refinement and *abstraction* are opposite and complementary types of relationships or design activities. Through refinement, an abstraction is made more concrete through the introduction of details, entailing design or implementation decisions, while through abstraction, details of a more concrete abstraction are omitted. An important property of refinement is that the resulting model should conform to the original one [1].

Design methodologies normally define different abstraction levels to be used for particular viewpoints. In these methodologies, abstraction levels are usually related to milestones in the design trajectory, or are related with particular design goals. Several design methodologies also define refinement (and abstraction) relations in order to guide the development of related abstraction levels.

2.2 Metamodelling

Metamodels can be used to define the syntax and semantics of models. When instances of the elements of a model *B* are used to produce a model *A*, *B* is said to be the *metamodel* of *A*. In this case, one can say that the *abstract syntax* of the model *A* is defined in the metamodel *B* [4]. Furthermore, model *A* can be considered as an instance of metamodel *B*.

The abstract syntax of a metamodel *B* can also be described in yet another metamodel *C*, thus constituting a metametamodel. Although the number of metalevels

is arbitrary, metamodelling frameworks should define a limited number of useful metalevels.

Whenever a metamodel is accompanied by natural language descriptions of concepts that correspond to its elements, we say that the *semantics* of the modelling elements are informally defined. This approach has been adopted by OMG in the Meta-Object Facility (MOF) [11] and in the UML proposed standards [16, 17]. More rigorous approaches define the semantics of modelling elements in terms of a mathematical domain (e.g., the formal semantics of the Specification and Description Language (SDL) in [8]), or in terms of concrete, formal and explicit representations of domain conceptualisations (e.g., an ontology [6]).

3 Model driven architecture

The MDA approach [13] to *system (application) specification*, *portability* and *interoperability* is based on the use of *formal* and *semi-formal models*. From the perspective of systems development, a significant quality of the MDA approach is the *independence* of system specifications (i.e., sets of models) from potential target implementation platforms. A system specification exists independently of any implementation platform and has formal or semi-formal transformation rules to many possible target platforms. The application development effort is consolidated in the platform-independent models, such that the investments necessary to move to another platform can be reduced. Furthermore, model transformation rules may be implemented in model-driven tools to (partially) automate the transformation of platform-independent models into platform-specific models, increasing the level of automation of the development trajectory.

From the perspective of systems interoperability, the use of platform-independent models facilitates the creation of different platform-specific models corresponding to the same set of platform-independent models, which results ultimately in implementations that can be easily (if not automatically) integrated.

Platform-independent models also play an important role in the re-use of legacy applications. In this case, integration is done at a platform-independent level, using platform-independent models that represent the legacy application. These platform-independent models are derived by reverse engineering.

3.1 MDA viewpoints

The MDA generally defines a platform as a set of subsystems or technologies that provide coherent functionality through interfaces and specified usage patterns. Any subsystem that depends on the platform can use this functionality without concern for the details of how it is implemented [13].

Three different viewpoints are considered [13]: computation-independent viewpoint, platform-independent viewpoint and platform specific viewpoint. The computation-independent viewpoint focuses on the system environment and its requirements. However, there is no concern for the details of the structure and

processing of the system. The platform-independent viewpoint focuses on the system operation, but hides the details necessary for a particular platform. The platform-specific viewpoint combines the platform independent viewpoint with the details of the use of a specific platform by a system.

A computation independent model (CIM) is a model developed according to the computation independent viewpoint. Similarly, a platform independent model (PIM) and a platform specific model (PSM) are models developed according to the platform independent and platform specific viewpoints, respectively.

Platform independence is a relative term that depends on the potential target platforms. For example, if the set of technologies that define a platform comprehends middleware platforms, such as, e.g., CORBA and Web Services, a CORBA Interface Definition Language (IDL) specification is a platform-specific model, because it is bound to CORBA. In contrast, if the set of technologies that define a platform comprehends programming languages and CORBA ORB implementations, such as, e.g., the C++ language and the C++ ORB implementation, a CORBA IDL specification is a platform-independent model, because it can be mapped onto several programming languages.

3.2 Model transformation

Model transformation is basically seen as a mapping of elements of one model onto elements of another model. Consider, for example, the creation of software systems by code generation. Each generated artefact, either some code in a programming language or some textual deployment artefact can be manipulated as a model. These models are based on a defined structure, which itself forms a metamodel. This metamodel can be expressed in terms of the UML and/or MOF standards.

Model transformation is useful if formally or systematically defined. As depicted in Figure 1, a transformation may be defined at the level of metamodels. When transformation is applied, a source model is transformed into a target model according to the defined transformation (rules).

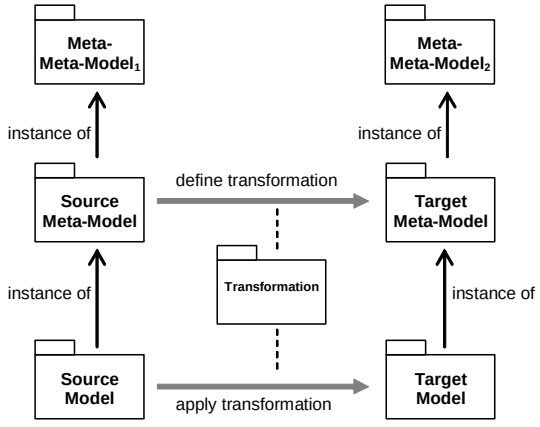

Figure 1: Model transformation with separate meta-metamodels

According to OMG definitions, a metamodel is based and constructed from elements of an underlying meta-metamodel (the MOF) and a model is constructed from elements of the metamodel. The use of a common meta-metamodel for the target and source metamodels, as illustrated in Figure 2 may facilitate the definition of transformations.

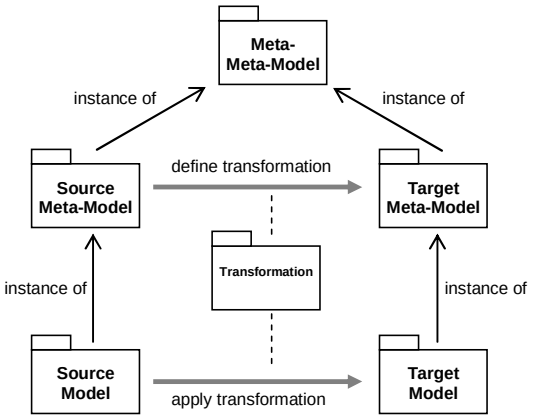

Figure 2: Model transformation with a common meta-metamodel

The model transformation pattern can be applied successively. In this case the notions of source and target models are relative. An intermediary model is considered a target model from the perspective of the transformation from the source model, and the same intermediary model is considered a source model from the perspective of the transformation to the final target model.

In order to allow a developer to guide the transformation of a source model when necessary, transformations may be parameterised. An annotation model may be used to hold the parameters for a transformation. The application of the transformation may include a step that transforms the source model into an annotated source model and then proceeds with the transformation.

4 Unified modeling language (UML)

This section presents an overview of the modelling capabilities in the Unified Modeling Language (UML), which has been standardized under the auspices of the Object Management Group (OMG). Our discussion is primarily based on UML 1.4 and 1.5 [14, 18] specifications. UML 1.5 is the currently adopted UML specification by OMG. However, most of the currently available UML tools provide support only to UML 1.4 specification. The UML 1.5 specification extends UML 1.4 with the so-called *action semantics*, which mainly adds more preciseness to the definition of actions and procedures. Since the first documents of UML 2.0 [16, 17] have just been publicly released, we do provide a highlight of the main changes in this specification with respect to the previous one.

4.1 Structure modelling

UML defines a collection of diagrams for structure modelling, namely class diagrams, component diagrams and deployment diagrams. UML does not prescribe how these diagrams should be used in a development trajectory, but only their abstract syntax and intended semantics (to a certain extent and informally). A development methodology should be applied in an actual development project to define the UML diagrams that have to be produced to represent a certain model at the different phases, steps or workflows devised for the development trajectory. Since component diagrams are the most relevant type of diagram for the development of service architecture, we exempt ourselves from discussing class and deployment diagrams in this paper.

A component diagram captures dependencies among different kinds of software components, such as implementation classes, source code files, binary code files, executable files, and scripts. A component diagram has only a type form, not an instance form.

UML 1.5 defines a component as a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component diagram is a graph of components connected by dependency relationships. Interfaces and calling dependencies among components can also be captured using a component diagram. A calling dependency occurs when a component uses a given interface. In such case, dependency arrows from components to the interface on other component must be employed.

A component diagram is used to model the static implementation view of a service. Thus, the architecture of a given service is captured as a collection of components and their dependency relationships. Since UML 1.5 does not consider a component as a unit of design, but a unit of deployment, the role of component diagrams in service development is rather limited. There is no support to component-based development, since UML 1.5 does not allow the representation of abstract components and does not support recursive decomposition of internal structures.

In UML 2.0, a component is a modular unit with well-defined interfaces, which allows it to be reusable and autonomous. The component concept has been introduced to support component-based development, in which components are modelled throughout the development trajectory, from abstract business components to concrete software components.

A component has one or more provided and required interfaces and its environment can only interact with it through these interfaces. The interfaces of a component shield the component's internal structure from its environment. Components can be composed together to form bigger components. This can be done by 'wiring' required interfaces to provided interfaces under the condition that these interfaces are compatible. This implies that a component *C* at some aggregation level is related to a collection of composed components that together realise component *C*.

A component can have required and provided interfaces: required interfaces are used by the component in order to perform its operation, while provided interfaces are those through which the component provides its capabilities. Required and provided interfaces are directly related to the direction of operation invocations, i.e., operations

at the required interfaces are invoked by the component itself, while operations at the provided interfaces are invoked by the component's environment.

Alternatively, designers may group interfaces in a port, which defines an interaction point between a component and its environment, or between a component and some elements of its internal structure. Ports allow an even stronger decoupling of a component from its environment than what is already possible using only interfaces. A component can be defined separately from its ports, making it reusable in any environment that complies with the constraints imposed by these ports. Ports also group interfaces, so that the possibly different aspects of the interactions with a component can be properly separated.

UML 2.0 allows the specification of the internal details of a component in some different alternative ways. Because a component is also a class, one can define inside a component all the other classifiers (e.g., components and classes) that are non-shareable parts the component. A more detailed representation of the internal structure of a component can be defined by showing instances of the classes owned by the component and how they relate to the component's ports.

4.2 Behaviour modelling

UML 1.5 defines a collection of diagrams for behaviour modelling, namely use case diagrams, sequence diagrams, collaboration diagrams, statechart diagrams and activity diagrams. Similarly to the structural diagrams, UML 1.5 does not prescribe how these diagrams should be used in a development trajectory. Although, use cases diagrams are considered behavioural diagrams, these diagrams can only capture static behaviour. Thus, we exempt ourselves from discussing it in the scope of this work.

Sequence diagrams

A sequence diagram shows how roles interact with each other in time, by showing the messages they exchange; alternatively, it may represent this by means of instances of the roles and the stimuli they produce. A sequence diagram can be seen as a set of messages and their temporal ordering. It relates to the system structure in that the roles and messages defined in a sequence diagram should correspond to classifiers and operations defined in structural diagrams.

A sequence diagram shows classifier roles (or instances), and the messages (or stimuli) they exchange. A sequence diagram represents either: (1) an interaction, consisting of message exchange between classifier roles and possibly the consequences of these messages (the actions), or (2) an interaction instance set, consisting of stimuli exchanged between instances of classifier roles and possibly the consequences of these stimuli (the actions). Sequence diagrams are also capable of representing other aspects of the systems dynamics, like object creation and destruction, conditional stimuli and focus of control.

Sequence diagrams define behaviour 'by example'. They are normally used to show how the system performs some specific parts of its functionality. They are also related to some scenario of execution or operation phase. Sequence diagrams can be useful at

a high abstraction level, when the designer wants to understand the global pattern of interaction between system parts, and at a low abstraction level, when details of the interaction between (more concrete) parts have to be described. Because they represent the partial behaviour of multiple system parts, they are suited as requirements for testing and verification, but are less suited for automatic code generation.

UML 2.0 introduces capabilities to define interaction fragments (smaller sequence diagrams) and to combine them together to form more complex sequence diagrams. These capabilities include (conditional) branching of interaction fragments, and references to interaction occurrences that can be defined separately. The gates concept has been introduced to connect different interaction fragments. The most important benefit of these new capabilities is the possibility of structuring sequence diagrams in terms of smaller fragments, increasing in this way the readability of sequence diagrams, mostly of importance in the case of complex diagrams. These capabilities make it possible to define behaviours more concisely and completely, approaching in this way the purpose and expressiveness of state charts.

Timing diagrams have been introduced in UML 2.0 to represent state changes and conditions on a timeline. Timing diagrams are expected to be useful for systems that have stringent timing constraints. UML 2.0 also defines the so-called interaction overview diagram, which allows sequence diagrams to be combined in the scope using the operator of activity diagrams. This implies that more alternative scenarios can be represented in a single diagram. Interaction overview diagrams bring interaction diagrams closer to activity diagrams (see section 4.2.3), by allowing them to represent behaviours in a more complete way.

Collaboration diagrams

The purpose of collaboration diagrams is similar to the purpose of the sequence diagrams (define behaviour through scenarios), but collaboration diagrams put more stress on the collaboration itself, i.e., on the roles participating in the collaborations and the associations between these roles. A collaboration diagram shows these roles and associations and plots an ordered set of directed message exchanges on the associations, in order to denote a specific interaction sequence.

A collaboration diagram shows classifier roles (or instances), their associations (or links), and the messages (or stimuli) they exchange. It contains the same information as sequence diagrams, but it represents the associations (links) explicitly.

Collaboration diagrams can play the same roles in the development trajectory as sequence diagrams. They only differ in that collaboration diagrams are not really suitable to represent complex interaction sequences, since the reader is forced to follow numbered messages throughout the diagrams to understand the sequences being described. The main benefit of collaboration diagrams is the combined representation of structural and behavioural aspects in a single diagram.

In UML 2.0 collaboration diagrams have been renamed to communication diagrams. They correspond to simple sequence diagrams, i.e., sequence diagrams that do not use the structuring capabilities that have been introduced in UML 2.0.

Activity diagrams

Activity diagrams in UML 1.5 are special cases of statechart diagrams. In activity diagrams one represents activities and their relationships, which allows one to define a behaviour that describes processes or workflows. The activities themselves may consist of actions, which are (smaller) tasks that are internal to the activity. Activity diagrams also include some capabilities to define decisions and merging of execution flows, and synchronisation states. Swim lanes can be used to partition an activity diagram, e.g., in terms of the roles that are responsible for the different activities.

An activity diagram represents an activity graph, which is a variant of a state machine. In activity graphs one defines activities (action states) and transitions triggered by the completion of these activities.

Most (software) development methodologies prescribe that business processes should be explicitly specified in the initial development steps. Particularly in the case of the Unified Process [9], these business processes can be used to specify the (behaviour of) use cases.

At the level of business models it is often necessary to model the processes that have to be performed by the business without necessarily assigning parts of these processes to some specific people, departments or software applications. These models allow business architects to reason about the procedural steps of these business processes, abstracting from how these steps are supposed to be performed. In the course of the implementation trajectory, choices have to be made concerning the allocation of these procedural steps to physical or logical entities.

Models of business processes normally consist of related activities that have to be performed in these processes. There are many alternative techniques that are suited for modelling activities and their relationships. In UML 1.5, activities can be modelled using activity diagrams.

In UML 2.0 activity diagrams are no longer a special case of statechart diagrams. Activity graphs in UML 2.0 have a semantics that is closer to Petri Nets, making them more suitable for the representation of business processes.

Statechart diagrams

Statecharts can be used to represent the behaviour of an object instance or other entities such as use cases, actors, subsystems, etc. Statecharts are used to define behaviour in terms of reactions to stimuli (discrete events). A statechart defines a collection of states and state moves, such that whenever a stimulus occurs, the behaviour performs some actions and transitions (state moves). Since behaviours defined using statechart diagrams can get rather complex in the case of complex behaviours, UML has some additional capabilities to structure these diagrams, like sub-states and sub-machines.

A statechart diagram represents a state machine. In essence a state machine is a graph that consists of states and state transitions triggered by events. A state may contain a list of internal transitions, which consist of internal actions or activities to be performed by the state machine while in this state. An event is some occurrence that may trigger a state transition. Events can be either the change of some Boolean value, the expiration of a timeout, an operation call or a signal. A transition is triggered by an

event. A (simple) transition is a relation between two states (*state1* and *state2*), and defines that whenever the state machine is in *state1* and the event that triggers the transition is processed, the state machine moves to *state2*. Only one event is evaluated at a time and it is either discarded if it does not trigger any transition, or it triggers only one transition (interleaving semantics). A transition is said to be *fired* whenever it is performed (terminology derived from Petri Nets).

During the development of a (software) system one has to define the logical parts (objects) of the system and specify their behaviour. Statechart diagrams allow one to specify completely the behaviour of the logical parts of a system, as opposed to the partial specification through interactions supported by sequence diagrams and collaboration diagrams. A criticism on statechart diagrams is that although it is suitable for the specification of behaviours that are relatively close to the implementation code, its interleaving semantics makes it less suitable to specify the behaviour of logical parts that may be decomposed and distributed, i.e., behaviours at higher abstraction levels. These more abstract behaviours can be useful for early analysis (e.g., through simulation) and since they constitute a statement on functional requirements, they can also be used for conformance assessment (verification).

In UML 2.0 interfaces can own a (protocol) state machine. Entry and exit points and terminate pseudo-states have been introduced to facilitate the reference to state machines and to improve structuring, allowing reusability of state machines. UML 2.0 allows a state list to be represented by a single state symbol. State machine extension allows one to reuse the definition of a state machine and extend it with additional states and transitions. A state machine may own other state machines, which can be referenced from its internal states.

Action semantics

In UML, an action is a fundamental unit of executable functionality. Until UML 1.4, actions in an activity could only be defined as strings, typically an action-expression added to a transition definition. This implies that no standard semantics existed for actions, which has been a major obstruction, amongst other, for the interchange of information between simulation tools [21].

The Action Semantics initiative has been started to define more precisely the meaning of actions in UML; results of this initiative have been incorporated already in the UML 1.5 specification [18]. In the UML 1.5 specification, a package Action has been added to define action semantics, with minor needs for readjustment in the rest of the language. UML 2.0 is built upon UML 1.5 for the behaviour part, i.e., the action semantics work done in UML 1.5 has been reused in UML 2.0. In the remaining of this section we refer to UML 2.0 and how it handles action semantics.

From an abstract point of view, an action is a fundamental unit of executable functionality in an activity that contains the action. The execution of an action implies a transformation or processing in the modelled system. An action may have sets of incoming and outgoing activity edges, through which it gets its input and delivers its output values, respectively. Incoming and outgoing edges define the control and data flows that determine whether an action is allowed to be performed. The completion of an action may enable the execution of other actions that depend on this action. Actions

may have pre- and post-conditions. Streaming parameters allow actions to start generating outputs while consuming inputs.

The different action types are used to define the semantics of individual actions. The following types of actions have been identified:

- Invocation actions: operation calls and the sending of signals, either to specific targets or broadcasting to potential targets;
- Read write actions: creation and destruction of objects, reading and modifying the values of variables and structural features (e.g., attributes) and creation and destruction of links;
- Computation actions: transformation of a set of input values to a set of output values by invoking a function.

Some additional read-write actions defined in UML 2.0 concern reading instances of a classifier, reading links of an association, determining the classifier of an instance and determining the association of a link. These specific actions allow one to specify system with introspective or reflexive capabilities, since they allow the behaviour to reason about the structure of the system itself. In order to completely define the semantics of events in a state machine, the acceptance of an event is defined as a specific form of action. The raising of an exception is also defined as a specific form of action.

The definition of action semantics makes it possible for tool vendors to develop tools for simulation and verification of behaviour specifications based on standard semantics. Action semantics has also enabled initiatives that aim at defining executable UML specifications, for simulation or even for prototyping. Executable UML [20] is an example of such an initiative.

4.3 Language extensibility

In the context of the MDA, an important characteristic of UML is its extensibility capabilities. UML is currently being positioned as a general purpose language that is expected to be customized for a variety of domains, platforms and methods [16].

A mechanism called profiling is used to enable lightweight extensions of the language. UML profiles extend the UML metamodel by specializing elements of the metamodel. The use of UML profiles enables the reuse of UML's notation and tools.

In case customization requirements exceed the capabilities offered by profiles, new languages may be defined via MOF metamodels. MOF 2.0 metamodels are being designed as instances of a subset of the UML 2.0, so that it will be possible to represent them using UML class diagrams.

5 Final remarks

Technologies such as Web Services and component models have been advocated as silver bullets for the service-oriented development of e-applications. However, if we are to draw any lessons from the past, then the most important would be: there are no

(lasting) one-fits-all technology solutions, and therefore technologies will always differ and evolve. Direct mappings onto whatever specific technology, independently of how popular this technology may be at a particular point in time, will lead to inflexible systems: it results in technology lock-ins and hinders interoperability, portability and integration when, inevitably, the technology changes or new technologies enter the scene. MDA has been introduced against this background. The MDA approach separates business (computation-independent) models, computation (platform-independent) models and platform models, enabling reuse of design artefacts at any of these levels, thereby providing possibilities for better return on investment. In addition, by defining transformations between these models, development projects can be done in a shorter time span and with higher quality. Not surprisingly, MDA led to a massive attention for models, modelling languages and transformations, primarily concentrated around UML.

UML has been developed for the specification of object-oriented systems and, as such, the concepts and abstractions used in UML facilitate the development of software using object-oriented implementation technologies. This yields a language that can potentially be translated into executable code or even executed directly, supporting the last stages of a model-driven development trajectory.

Despite that UML behaviour specification capabilities may be suitable to support final stages of a model-driven development trajectory, the use of UML for behaviour specification in earlier stages of a model-driven development trajectory should be further investigated. Recently, constructs for component-based development have been incorporated into UML 2.0, allowing the concept of component to be recursively applied through the development trajectory. Nevertheless, the refinement of behaviour specifications is a deficiency of UML's behaviour representation capabilities. Still lacking is a notion of behaviour conformance in order to relate behaviours defined at a high-level of abstraction and the refined realizations of these behaviours [4]. Consequently, we cannot formally assess the correctness of component compositions. Activity diagrams used to express behaviour in an integrated perspective, e.g., for the purpose of business modelling, are not related by refinement to statecharts that distribute responsibilities of a business process to specific services and components that support this business process.

In the context of MDA, much effort has been invested in language definition and extension mechanisms, metamodelling, model transformation specification and tool support. The study of platform-independence, however, has been somewhat overlooked. We have been striving to address this in our research [1, 2], by providing guidelines for the selection of abstraction criteria and modeling concepts for platform-independent modeling. Further research is necessary in order to define criteria to ensure the beneficial exploitation of the PIM-PSM separation of concerns adopted by MDA.

References

1. Almeida, J.P.A., Sinderen, M. van, Ferreira Pires, L. and Quartel, D.: A systematic approach to platform-independent design based on the service concept. In: *7th IEEE Intl. Enterprise Distributed Object Computing (EDOC) Conference*, Sept. 2003, pp. 112-123.
2. Almeida, J.P.A., Sinderen, M. van, Ferreira Pires, L. and Wegdam, M.: Handling QoS in MDA: a discussion on availability and dynamic reconfiguration. In: *Workshop on Model Driven Architecture: Foundations and Application (MDAFA) 2003*, CTIT Technical Report TR–CTIT–03–27, University of Twente, The Netherlands, June 2003, 91-96.
3. De Farias, C.R.G.: *Architectural Design of Groupware Systems: a Component-Based Approach*. PhD thesis, University of Twente, Enschede, the Netherlands, 2002.
4. Dijkman, R.M., Quartel, D., Ferreira Pires, L. and Sinderen, M. van: An Approach to Relate Viewpoints and Modeling Languages. In: *7th IEEE Enterprise Distributed Object Computing (EDOC) Conference*, Sept. 2003, pp. 14-27.
5. Harel, D. and Rumpe, B.: *Modelling Languages: Syntax, Semantics and All That Stuff*. Technical Report, The Weizmann Institute of Science, Rehovot, Israel, MCS00-16, 2000.
6. Guizzardi, G., Ferreira Pires, L. and van Sinderen, M.: On the role of Domain Ontologies in the Design of Domain-Specific Visual Languages. In: *2nd Workshop on Domain-Specific Visual Languages, ACM OOPSLA*, 2002.
7. International Telecommunications Union (ITU): *Open Distributed Processing Reference Model. Part 1 - Overview*. ITU Rec. X.901 | ISO/IEC 10746-1, Geneva, 1997.
8. International Telecommunications Union (ITU): *SDL Formal Semantics Definition*. ITU Rec. Z.100, Annex F, Geneva, 2000.
9. Jacobson, I., Booch, G. and Rumbaugh, J.: *The unified software development process*. Addison Wesley, USA, 1999.
10. Object Management Group: *MDA Guide Version 1.0*. May 2003.
11. Object Management Group: *Meta Object Facility (MOF) 2.0 Core Proposal*. April 2003.
12. Object Management Group: *Meta Object Facility (MOF) Specification 1.4*. April 2002.
13. Object Management Group: *Model Driven Architecture (MDA)*. July 2001.
14. Object Management Group: *OMG Unified Modelling Language Specification, version 1.4*. September 2001.
15. Object Management Group: *OMG XML Metadata Interchange (XMI) Specification*, Version 1.2, January 2002.
16. Object Management Group: *UML 2.0 Infrastructure Specification*. September 2003.
17. Object Management Group: *UML 2.0 Superstructure Specification*. August 2003.
18. Object Management Group: *Unified Modeling Language Specification, version 1.5*. March 2003.
19. Quartel, D.A.C.: *Action relations. Basic design concepts for behaviour modelling and refinement*. CTIT Ph.D-thesis series, no. 98-18, University of Twente, Enschede, The Netherlands, 1998.
20. Mellor, S.J. and Balcer, M.J.: *Executable UML. A foundation for the Model-Driven Architecture*. Addison-Wesley, 2002.
21. Mellor, S.J., Tockey, S., Arthaud, R. and Leblanc, P.: *Software-platform-independent precise action specification for UML*. White paper. http://www.projtech.com.