Delft University of Technology
Software Engineering Research Group
Technical Report Series

Visualisation of Domain-Specific Modelling Languages Using UML

Bas Graaf and Arie van Deursen

Report TUD-SERG-2006-019a

TUDelft

SERG

TUD-SERG-2006-019a

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS), Tucson, AZ, U.S.A, March 2007.

© copyright 2006, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Visualisation of Domain-Specific Modelling Languages Using UML

Bas Graaf
Delft University of Technology
The Netherlands
b.s.graaf@tudelft.nl

Arie van Deursen
Delft University of Technology and CWI
The Netherlands
arie.vandeursen@tudelft.nl

Abstract

Currently, general-purpose modelling tools are often only used to draw diagrams for the documentation. The introduction of model-driven software development approaches involves the definition of domain-specific modelling languages that allow code generation. Although graphical representations of the involved models are important for documentation, the development of required visualisations and editors is cumbersome. In this paper we propose to extend the typical model-driven approach with the automatic generation of diagrams for documentation. We illustrate the approach using the Model Driven Architecture in the domains of software architecture and control systems.

1. Introduction

Model-driven engineering (MDE) refers to software development approaches in which models are considered the primary development artefacts [1] (instead of source code). In these approaches software models are gradually transformed (automatically) into source code by means of model transformations. Additionally, such models are used for other (automated) software engineering tasks, such as performance analysis.

Typically, MDE approaches are based on modelling languages that offer abstractions focussed on a particular domain. Such languages are referred to as domain-specific modelling languages (DSMLs). From DSML-models code is generated for a particular software platform. DSMLs have been developed for various types of domains, such as software engineering (e.g., software architecture [2]) and application domains (e.g., insurance products [3]).

In general, the use of DSMLs has clear advantages over the use of general-purpose languages [4]. More in particular, in the context of MDE, our experience in industrial case studies [5, 6] indicates that the use of a general-purpose language, such as UML, leads to (unnecessary) complex model transformations, for instance to generate code. As such, the introduction of MDE typically requires the development of DSMLs.

Although mechanisms are available to define and implement the abstract syntax of DSMLs, such as the MetaObject Facility (MOF [7]) and the Eclipse Modeling Framework (EMF [8]), not much support is available for the definition their graphical notation (concrete syntax). As a result development of adequate graphical editors and visualisations requires considerable effort.

For some software engineering tasks, such editors are not required. For instance, developers can use a textual syntax for the creation of models that can subsequently be processed by model transformation tools. However, other tasks, such as documentation, do require some form of graphical representation. It is this problem that motivates this paper.

The basic idea of this paper is simple: when devising a new DSML we try to leverage existing visual notations and modeling tools. We propose to expand the typical MDE process in which abstract models are gradually transformed into code, with (partial) generation of documentation. To this end we combine the use of DSMLs for code generation and other automated software engineerings tasks, with the use of UML for documentation. The approach uses model transformations to specify the mapping between DSMLs and UML. The diagrams corresponding to those models, as visualised by off-the-shelf UML tools, are used in the documentation. To investigate the arguments for and against this idea, we study how

- this approach works for various architectural views;
- UML can be used as the target language for visualising these views; and
- model transformations can be used to specify and automate the mapping.

In practice, the extra effort required for the development of graphical editors can hamper the introduction of MDE. Consider the following scenario. A software development organisation decides to introduce MDE. Currently, the developers use UML. However, as in many other organisations, they only use UML modelling tools for *drawing* diagrams [9]. These diagrams are important for the communication with other stakeholders, as they

constitute an essential part of the documentation. The introduction of MDE involves the definition of DSMLs from which code will be generated. Furthermore, as the developers are comfortable with using a textual syntax for these DSMLs, no graphical editors are developed. The result is that they now have to create DSML models for code generation as well as UML diagrams for documentation. Considering the current use of UML, as investigated by Lange et al. [9], and the upcoming of MDE approaches, such as the Model Driven Architecture (MDA, [10]), this is not an unlikely scenario.

We investigate the feasibility of our approach in the domain of software architecture. In Section 2 we introduce the languages specific to this domain, and the standard documentation approach. Our approach for the model-driven documentation of software architecture, MDAV, is presented in Section 3 and we report on a small case study in Section 4. The approach is easily applied to other domains as well. An additional (industrial) case study involving a different type of models is presented in Section 5. We discuss the benefits and limitations of the approach in Section 6. After discussing some related work in Section 7, we conclude with an overview of our contributions and opportunities for future work in Section 8.

2. Background

In this section we introduce modelling and documentation in the domain of software architecture. Furthermore, we discuss some of the technologies that enable our approach.

2.1. Software architecture

Modelling Several notations have been developed to specify architectural models. These architecture description languages (ADLs, see [2] for an overview) mostly consider an architecture to be a configuration of runtime components and connectors.

Due to their formal syntax and semantics ADLs enable automatic code generation and analysis. Despite these benefits, and although ADLs have received much attention from the architecture research community, they have not been applied much in industry [11].

Although UML is aimed at object-oriented modelling, it allows practitioners to address a wide range of issues [12]. Therefore, and because of the availability of supporting (graphical) modelling tools, it is often used in practice to describe software architectures [13, 9]).

A drawback UML is the semantic mismatch between architectural and UML's OO concepts, resulting in compromises between completeness and legibility [14]. Furthermore, for automatic processing of models (e.g., for code generation) the complexity of UML results in complex model transformations [5, 6].

Documentation Because in industrial practice a software architecture is too complex to describe in a single stroke different views are used for its documentation. Different types of views have been defined to address specific concerns. The two most prominent categories of architectural views are module views and component-and-connector (C&C) views [15].

A module view addresses the question of how a system is *developed*; it defines the most important implementation units (modules) and their relations. Module views are used, for instance, to evaluate the maintainability of a system as implied by its architecture.

A component-and-connector (C&C) view, on the other hand, addresses the question of how a system *works*. It describes a system in terms of runtime components and connectors. A component is an abstraction of a computational element; a connectors is an abstraction for the way components interact. As such, a C&C view is more suited for analysis of runtime properties, such as performance.

More specific types of views are defined by imposing restrictions on the type of elements and relations allowed in a view. In a module-uses view, for instance, only 'uses' relations are allowed.

In the terminology of IEEE Std 1471-2000 [16], a view conforms to a viewpoint that "specifies the conventions for using and constructing a view". A viewpoint addresses a set of stakeholder concerns. A number of viewpoint sets is available from literature, such as [15]. Furthermore, in practice also custom viewpoints are defined. Typically, a viewpoint definition prescribes a modelling language or notation that enables the specification of an architectural model that addresses the concerns of the viewpoint. As an example, a C&C viewpoint might refer to a particular ADL. In summary, a viewpoint defines a type of views and a view is a particular representation of a particular system.

In practice the architectural model for a view is primarily used as a figure or diagram (the view's primary presentation [15]) in a document that describes the view. Because of their wide-acceptance and available tool support often UML diagrams are used for this [13].

2.2. Enabling MDE technologies

Our approach for model-driven documentation is based on model transformations. This requires capabilities for (meta)modelling, model transformation, and model interchange.

For the definition of metamodels we use the MetaObject Facility (MOF [7]). An implementation of

the MOF is available as an Eclipse plugin, the Eclipse Modeling Framework (EMF [8]).

The EMF plugin generates an implementation for a metamodel as a set of Java classes that offers an interface that allows developers to manipulate conforming models. These models can be serialised to an XML document using XML Model Interchange (XMI [17]). Additionally, a simple tree-based editor is generated that can be used as an Eclipse plugin for the creation and inspection of associated models. As an example consider the screenshot of such an editor in Figure 3(b). This editor is also capable of validating a model against its metamodel.

The Atlas Transformation Language (ATL [18]) is based on EMF. We use it to define model transformations that are executed by a transformation engine. In ATL, transformations are defined in modules that consist of declarative transformation rules and helper operations. Using a syntax similar to that of OCL, the transformation rules match model elements in a source model and create elements in a target model. A helper is defined in the context of a metamodel element, to which it effectively adds a feature.

For their input, model transformation tools typically use XMI serialisations of MOF-based (meta)models. In the case of UML, these models can simply be created and visualised using standard UML tooling.

3. Model-Driven Architectural Views

To take advantage of the power of DSMLs for code generation and other automated software engineering tasks and that of UML for documentation, we explicitly distinguish architectural documentation and architectural models. We make this concrete by revisiting the conceptual model of the industry standard for description of software architectures (IEEE Std 1471-2000 [16]). The result, the Model-Driven Architectural Views (MDAV) framework, is displayed in Figure 1.

3.1. MDAV Framework

In Figure 1, for the development of a software System, an Architecture is defined that includes the most important design decisions. These are made concrete in an Architectural Description that consists of Models on the one hand, and architectural Views on the other. In the spirit of MDE, models conform to a Metamodel and are used for several (automated) tasks such as, analysis and code generation. Views on the other hand conform to a Viewpoint and are primarily used for communication purposes. Both metamodels and viewpoints are developed to address a certain set of Concerns. A viewpoint prescribes the language to be used to model the architecture. A metamodel specifies the abstract syntax of this language.

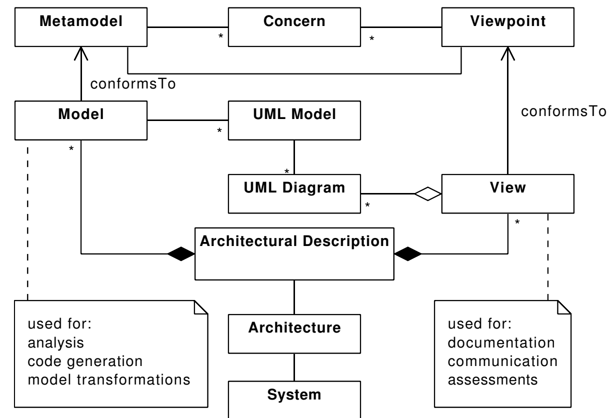


Figure 1. MDAV framework

A view includes diagrams in its primary presentation that represent the associated architectural models.

To allow the use of custom defined DSMLs without the need to specifically develop corresponding graphical representations and editors, we use UML Diagrams. To this end, we map DSML Models to UML Models that are visualised as UML diagrams for inclusion in view documentation with standard UML tooling. Thus, in MDAV the connection between views and models is made through (UML) diagrams. Thanks to this connection, views become model driven.

3.2. MDAV Process

In summary, compared to the conceptual model as described by IEEE Std 1471-2000, we add the concept of a diagram that allows to relate a view to a model. Furthermore, in-line with MDE, we explicitly added a metamodel as a description of the modelling language used in a view. Application of the corresponding approach involves three steps: definition of 1) a suitable metamodel, 2) means to create corresponding models, and 3) a mapping to UML.

A suitable metamodel for a particular viewpoint can be defined from scratch or based on an existing ADL that addresses the relevant concern. In the former case, we use a description of the viewpoint (e.g., from Clements et al. [15]) and create corresponding elements and relations in the metamodel. In the latter case, we base the metamodel on the ADL's grammar (or other language specification mechanism). Given the typically modest size and simple syntax of ADLs and using appropriate tooling, corresponding metamodels are easily created.

A means to create models associated with the defined metamodel is also required. Depending on the complexity of the associated metamodel different alternatives are suitable, of which we give examples in Section 4.

We specify and implement the mapping between the

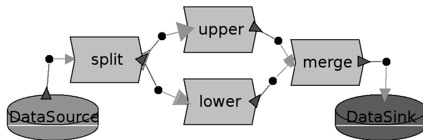


Figure 2. C&C model of CaPiTaLiZe (ACME)

prescribed metamodel and UML using a model transformation language. For several ADLs mappings to UML already exist, that we can specify as model transformations. This allows us to automatically transform an architectural (ADL) model to a UML Model. As such, ADL Models and UML Diagrams can evolve simultaneously.

Although the corresponding UML diagram might not exactly represent the architectural model (e.g., because the latter uses concepts that do not correspond to any UML concept), it is typically complete enough for most communication purposes. Moreover, in the case of a semantic mismatch, we use stereotypes to indicate the type of ADL element a specific UML element represents.

4. Using MDAV to Generate Views

We applied MDAV to two architectural viewpoints: we defined an appropriate metamodel, that is, a modelling 'language', means to create and manipulate associated models, and a mapping to UML,

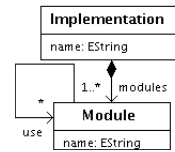
We use the CaPiTaLiZe system [19] as a running example. CaPiTaLiZe transforms a character stream by capitalising alternate characters. A C&C model of CaPiTaLiZe defined using an ADL (ACME [20]) is visualised in Figure 2. CaPiTaLiZe is designed as a pipe-and-filter system, with separate components for splitting a stream of characters in two streams (split), (un)capitalising characters (upper, lower), and merging two streams of characters (merge).

The diagram of CaPiTaLiZe's module view is depicted in Figure 3(c). In this UML class diagram we represent architectural modules with UML Packages and use-relations with UML Dependencies, as suggested by Clements et al. [15].

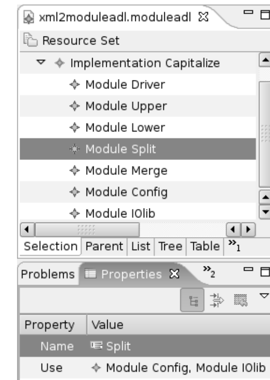
4.1. Module-uses view

Metamodel Module-uses views are based on a special type of dependency relation: the uses relation. As such, these views only contain one type of element and one type of relation [15].

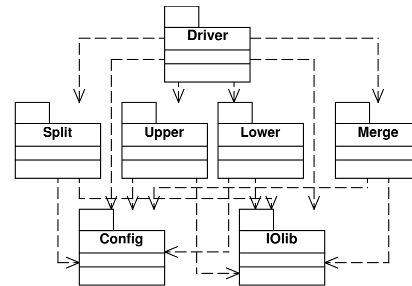
Although UML is well-suited and therefore also typically used in the primary presentation of module views, we developed a small custom metamodel to illustrate MDAV. This MODULEADL metamodel is depicted in Figure 3(a). In addition to a Module element and use relation it defines an Implementation to consist of a set of modules that may use other modules.



(a) MODULEADL metamodel



(b) MODULEADL model of CaPiTaLiZe in EMF editor



(c) MODULEADL diagram (UML)

Figure 3. MODULEADL

Model creation Using MOF, in principle, only the abstract syntax is defined. Although XMI offers an off-the-shelf mapping to XML, it is not intended to be used directly by software developers [21].

For simple metamodels, such as our MODULEADL, we propose to use the editor generated by EMF for the creation and inspection of models. Figure 3(b) shows a screenshot of this editor while editing the MODULEADL model for the CaPiTaLiZe system. The top part shows the modules that are defined for this system, the Properties pane is used to inspect the properties of those modules. This screenshot shows, for instance, that Module Split uses Module Config and Module IOlib.

UML mapping The mapping to UML is based on one of the mappings suggested in [15]. We map Modules to UML Packages and the uses relation to UML Dependencies. We specified this mapping using ATL. A fragment is depicted in Listing 1.

In an ATL transformation rule a **from** clause specifies a pattern that is matched by elements of the source model. For each match the target patterns in the **to** clause are instantiated in the target model. In this case, the Package rule creates a Package (p) and a set of Dependencies (ds) for each Module (m) in the source model. Using the **distinct** ... **foreach** construct a Dependency is created for every Module that is used by the Module that matched the rule (m.use). For both target elements a set of bindings is

```
rule Package {
  from m:MADL!Module
  to p:UML!Package (
    name <− m.name,
    clientDependency <− ds),
  ds: distinct UML!Dependency foreach (um in m.use)(
    client <− m,
    supplier <− um)
}
```

Listing 1. MODULEADL → UML (ATL)

specified to initialise their features. The clientDependency feature of the created Package (p), for instance, is initialised with the set of Dependencies created by this rule as well (ds).

The result of applying this transformation to the MODULEADL model of the CaPiTaLiZe system (Fig. 3(b)), is visualised using a UML tool (Fig. 3(c)).

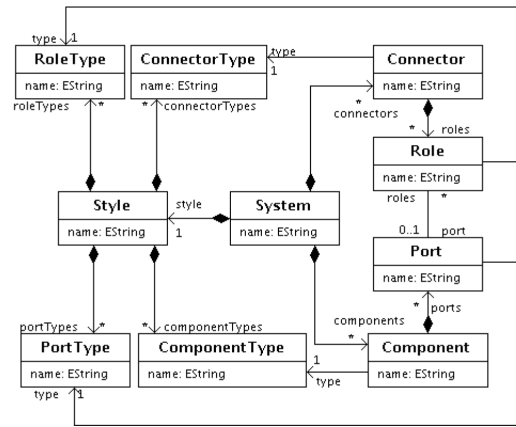
4.2. Component-and-connector view

Metamodel For C&C views, we define a metamodel for a simple ADL similar to ACME [20], an ADL interchange language that covers the most constructs in a wide range of ADLs.

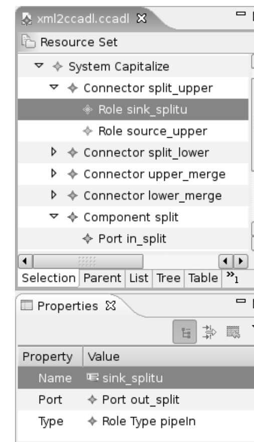
Consider the metamodel for the ADL (CCADL) in Figure 4(a). Using CCADL the architecture of a System consists of a Style, a set of Components, and a set of Connectors. A component owns a set of Ports via which it interacts with its environment. Similarly a connector owns a set of Roles that define what behaviour is expected from the participants in the interaction the connector represents. By attaching conforming roles and ports, configurations of components and connectors can be created. Finally, the style defines the types of components (ComponentType), connectors (ConnectorType), roles (RoleType), and ports (PortType).

Model creation Again, a straightforward approach to create CCADL models is to use the editor generated by EMF. Figure 4(b) displays a screenshot of this editor, while editing the CaPiTaLiZe CCADL model. When considering the complexity of the CCADL metamodel (compared to the MODULEADL metamodel), it becomes clear that editing models using this editor is inconvenient. Using this editor it is not possible, for instance, to immediately determine the component and connector types and understand their configuration.

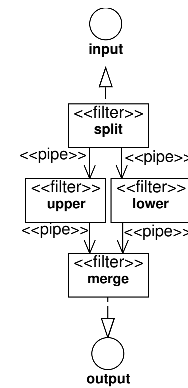
As an alternative, we propose to use a simple XML DTD or schema that allows to describe associated models as simple as possible. A fragment of an XML document conforming to such a DTD describing the same CaPiTaLiZe system is depicted in Listing 2. Note that, the DTD



(a) CCADL metamodel



(b) CCADL model of CaPiTaLiZe in EMF editor



(c) CCADL diagram (UML)

Figure 4. CCADL

we defined allows to separately specify the configuration of components and connectors as a set of attachments.

If we use simple XML documents to specify systems in CCADL, we separately need to populate a *model* conforming to the CCADL metamodel. Several approaches can be used to populate a model.

One possibility is to develop a so-called injector, a program that parses a file and uses the API generated by EMF to instantiate a corresponding model. In general, an injector is used to bridge two different domains, in this case the XML and modelware (MOF) domains. In the context of model-driven engineering such domains are also referred to as Technological Spaces [22].

As an alternative we reuse the XML injector, and XML metamodel (Fig. 5) provided by the ATL project[1]. Based on an XML document this injector instantiates a model that conforms to the XML metamodel. Subsequently, we transform this model into a model that conforms to the CCADL metamodel using ATL model transformations. The latter approach requires the smallest ef-

[1] `http://www.eclipse.org/m2m/atl`

```
...
<System name="Capitalize">
  <Style name="pf">
    <ComponentType name="Filter"/>
    <PortType name="filterOut"/>
    ...
  </Style>
  <Component name="split" type="Filter">
    <Port name="in_split" type="filterIn"/>
    <Port name="out_split" type="filterOut"/>
  </Component>
  ...
  <Connector name="split_upper" type="Pipe">
    <Role name="sink_splitu" type="pipeIn"/>
    <Role name="source_upper" type="pipeOut"/>
  </Connector>
  ...
  <Configuration>
    <Attach port="out_split" role="sink_splitu"/>
    ...
  </Configuration>
</System>
```

Listing 2. C&C model of CaPiTaLiZe (XML)

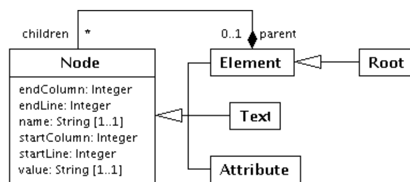


Figure 5. XML metamodel

fort because it reuses existing injectors and metamodels, and only requires us to specify a transformation that maps the XML metamodel to our CCADL metamodel.

The transformation to instantiate a CCADL model based on a (injected) XML source model is straightforward. Listing 3 contains a fragment of this transformation. The rule matches all XML elements named 'Component'. For each it creates a Component in the CCADL model. The type and name features are initialised using two helpers, getType and getName. They navigate the XML model to extract the desired information. For the type feature this is another XML Element that, in turn, matches a rule that creates ComponentTypes. The other elements of the CCADL metamodel are instantiated by similar rules.

UML mapping The UML representation of components and connectors is based on the strategies for modelling architecture with UML described by Garlan et al. [14]. In Figure 4(c) component and connector types are depicted as stereotypes, components as classes, connectors as associations, and ports and roles are not explicitly shown at all.

```
rule Component {
  from el:XML!Element(
    el.name='Component')
  to c:CCADL!Component(
    type <− el.getType,
    ports <− el.children−>select(e|e.name='Port'),
    name <− el.getName)
}
```

Listing 3. XML → CCADL (ATL)

```
rule Association {
  from conn:CCADL!Connector
  to asoc:UML!Association(
    name <− '<<' + conn.type.name + '>>',
    connection <−Set{conn.roles−>asSequence()−>first(),
      conn.roles−>asSequence()−>last()})
}
rule AssociationEnd {
  from r:CCADL!Role
  to aend:UML!AssociationEnd(
    isNavigable <− r.type.name='pipeOut',
    participant <− CCADL!Component−>allInstances()−>
      select(c|c.ports−>includes(r.port)))
}
```

Listing 4. CCADL → UML (ATL)

Listing 4 shows two rules of the corresponding ATL model transformation. The Association rule instantiates an Association (asoc) for each Connector (conn) in the source model. As our UML tool did not support stereotypes on Associations, we initialise the name feature to mimic one. Although not very elegant, this is acceptable when considering the goal of our transformation: generation of diagrams for documentation. In UML an Association has a connection feature that is a set of AssociationEnds. In our case, these represent the Roles of a Connector. For simplicity we assumed a Connector has exactly two Roles. The connection feature is initialised to the result of the rule that matches the roles of the Connector. Roles are matched by the AssociationEnd rule that creates an AssociationEnd (aend) for every matched Role (r). The isNaviagable feature is initialised depending on whether the matched Role is of type pipeOut (true) or not (false). As such, we control the direction of the Association for representation of the Connector.

Depending on the exact concerns the associated viewpoint addresses, alternative mappings to UML can be implemented similarly, such as a mapping that explicitly shows ports and roles.

5. Industrial Application

In this section we discuss a case study in which we applied MDAV to an architectural view in use for a class of control systems. Before discussing the three steps of our approach, we briefly introduce the case study.

ASML, a large manufacturer of equipment for the semi-conductor industry, studies the migration to a new architecture for supervisory machine control (SMC) components. In an advanced manufacturing machine, such as the wafer-scanners developed by ASML, an SMC component is responsible for the coordination of manufacturing activities in order to perform manufacturing requests. In a layered control architecture, an SMC component receives manufacturing requests from components in a higher layer, and coordinates the execution of manufacturing activities by components in a lower layer. Traditionally, the design for SMC systems is based on state transition models. The new approach [23] is based on task-resource models.

Metamodel Using the task-resource approach, SMC systems consist for a large part of generic, reusable components defined by a product-line architecture. The remaining application-specific components are generated based on a model of an SMC system in terms of tasks and resources. The associated metamodel is shown in Figure 6(a).

Task-resource models consist of a static and a dynamic part. The static part models the controlled System by specifying the Behaviours (manufacturing activities) it can perform, the Capabilities this requires, and the Resources (subsystems) it controls to offer those capabilities. The dynamic part models the manufacturing requests the component can perform in terms of (simple, conditional, or compound) Tasks that are of a specific Behaviour. Precedence relations between tasks are used to specify restrictions on execution order.

Based on the metamodel, tools can be developed for the generation of source code, model validation, and other software engineering tasks that can be automated. We used it, for instance, as the target of a model transformation that automates the migration of SMC components from a state-based to a task-resource-based architecture [6].

Model creation In this case, task-resource models were obtained by the automatic migration of legacy SMC models (based on state machines) to models based on the task-resource architecture. As such, a means to create task-resource models directly was not yet required. When SMC systems are developed based on the task-resource approach from scratch, such means would be required. In that case, one of the alternatives presented in the previous section can be used.

An example of a generated task-resource model (as result of the automated migration) inspected using the EMF editor is depicted in Figure 6(b). This editor was generated based on the metamodel we defined (Fig. 6(a)). Apart from this editor there was no (more advanced) editor available for these models.

UML mapping For the documentation of SMC systems based on the task-resource architecture, a viewpoint was defined. Due to the lack of a convenient editor to visualise task-resource models and to take advantage of available tooling and experience, the viewpoint prescribes that such models are depicted using UML diagrams. As such, the alignment of the task-resource view documentation with the task-resource models from which code can be generated, involved a mapping of the corresponding metamodel to UML.

For the documentation of a conforming view, separate diagrams are used for the static part and for each of the possible requests of the dynamic part. For the former, a UML Class Diagram is used in which a Class with appropriate Stereotype is used to represent a Behaviour, Capability, or Resource. For the latter, UML Activity Diagrams (one for each request) are used that effectively represent tasks and their precedence relations as task graphs. We used ATL to define corresponding mappings from the task-resource metamodel to UML class models for the static part, and to UML activity graphs for the dynamic part. A UML tool visualises these models as a UML Class Diagram, and a UML Activity Diagram, respectively.

As an example, the rule in Listing 5 maps a SimpleTask to the element that represented an activity in a UML Activity Diagram: ActionState. The name feature of the generated ActionState (state) is initialised using the name of the behaviour associated with the SimpleTask (st) that matched the rule. The rule also creates a set of Stereotypes (stype). In that target element the sTypes helper determines the resources required by the behaviour associated with the matched SimpleTask. This set is used to generate a set of ActionState Stereotypes used to initialise the stereotype feature of the generated ActionState.

Application of the transformation we defined to the model partly depicted in Figure 6(b), results in a class model and an activity graph for each request. One of those is visualised as an Activity Diagram in Figure 6(c). Tasks are represented by Activities, required resources by Stereotypes on Activities, precedence relations between Tasks by the order of the Activities, fork bars were used to indicate tasks that can be executed concurrently (i.e., tasks without precedence relations), and conditional Tasks (OrTasks) were mapped to choice nodes. As such, to complete the migration, models as the one depicted

```
rule ActionState {
  from st:TRS!SimpleTask
  to state:UML!ActionState (
    name <− st.behaviour.name,
    stereotype <− stype),
  stype: distinct UML!Stereotype foreach (s in st.sTypes)(
    name <− s,
    baseClass <− 'ActionState'),
  ...
}
```

Listing 5. TRS → UML (ATL)

in Figure 6(b) are used for model-based generation of source code, while diagrams as the one in Figure 6(c) are used for view-based *documentation*. Using model transformations the diagrams for this documentation are generated automatically.

6. Discussion

Our approach has several benefits. It reduces the effort required for the introduction of MDE approaches by circumventing the need to specifically develop graphical editors for the visualisation of DSMLs models. Furthermore it allows to introduce an MDE approach gradually; UML diagrams can continue to be used for documentation purposes. As such, in the case of software architecture, it facilitates the integration of ADLs and supporting tools in industrial development processes.

As presented here the approach uses MDA technology for model transformations and metamodelling. The underlying ideas are applicable to other MDE approaches as well: either by using the available transformation and metamodelling technologies for that MDE approach, or by implementing a bridge to MDA. We gave an example of the latter in Section 4.2 for XML.

Of course, the diagrams that are generated automatically using our approach, only constitute a minor part of the complete documentation. Architectural views, for instance, typically also document (some of) the rationale and trade-offs that underly design decisions [15]. In fact, an architectural view can be seen as 'diagrams + explaining text'. Although the 'explaining text' is not automatically updated using our approach, it does provides a starting point for doing so (i.e., the newly generated diagram).

Whether a mapping to UML is feasible, depends on the type of models involved and the documentation requirements. A potential risk of our use of UML, is that the UML semantics might not match with the semantics of the represented (DSML) model elements, resulting in ambiguities. In these cases appropriate stereotypes should be introduced. As an example, consider the stereotypes in Figure 4(c). These stereotypes are included in the ATL mappings we defined.

In the case that the semantic gap between the involved metamodel and UML is too large to be solved with stereotypes, instead of UML, more generic graph languages such as DOT[2] and GXL[3] could be used as target of the mapping.

The effort required for specification of the mappings to UML is mainly determined by the complexity and size of the DSML metamodel. Typically, these are relatively small (e.g., compared to UML). Furthermore, such mappings can be either specifically developed (as in the case of task-resource models) or reused (as in the case of ADLs). In the latter case they only need to be specified as a model transformation.

Our approach focusses on *visualisation* of DSML models. It does not offer visual *editing* for models conforming to complex metamodels. When that is required, editors have to be developed specifically. Technology to partly generate such editors is provided in the Eclipse Graphical Modeling Framework (GMF [24]) using EMF. Based on the specification of a concrete syntax and the abstract syntax specified by a metamodel this plugin can generate an editor. However, in the case that only visualisation is required, our approach offers a lightweight alternative.

Another alternative is to simply manually create documentation instead of automatically as in our approach. In that case the diagrams corresponding to some software models are created (drawn) manually using modelling or more generic tools. Obviously, consistency becomes an issue with such an approach.

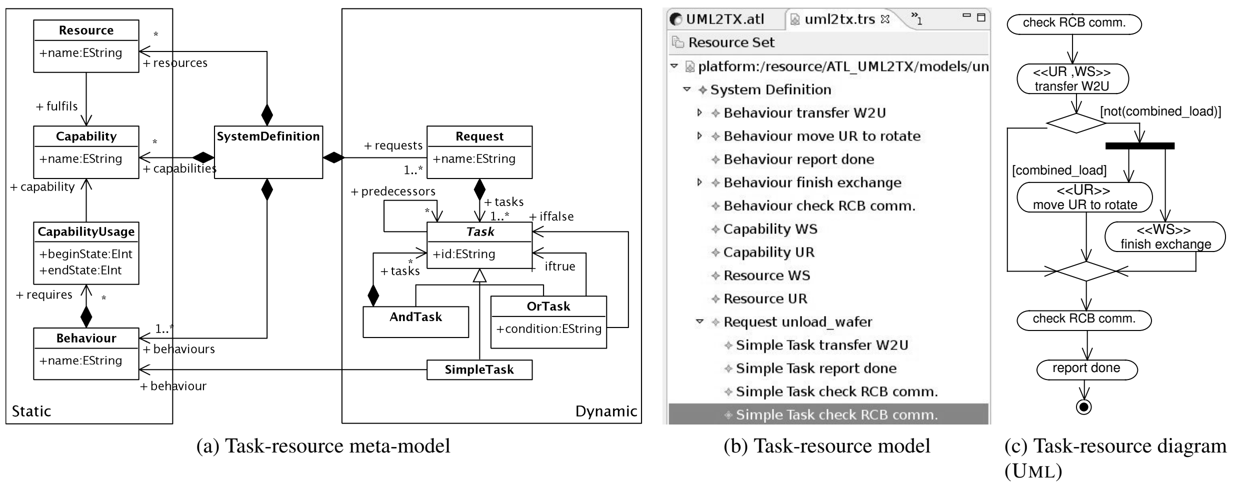
7. Related Work

Fondement and Baar [25] present an approach to specify (graphical) concrete syntax by extending metamodels. Based on this approach tools could be developed to (partly) generate corresponding editors. Instead, we take advantage of existing UML tools.

Medvidovic et al. [12] investigate the use of UML in the domain of software architecture. More in particular, they investigate how modelling constructs used in ADLs (i.e., a type of DSML) can be represented using UML. They consider two approaches for using UML to model software architectures: (1) use UML 'as-is', and (2) use UML's extension mechanisms. They conclude that UML has a number of limitations when used to model software architectures. The lack of architectural modelling constructs makes it necessary to adopt specific interpre-

[2] DOT - Language used by Graphviz (Graph Visualisation Software), see http://www.graphviz.org

[3] GXL - Graph eXchange Language, see http://www.gupro.de/GXL



(a) Task-resource meta-model (b) Task-resource model (c) Task-resource diagram (UML)

Figure 6. Task-resource metamodel, model, and UML representation

tations of UML model elements or to rely on the Object Constraint Language (OCL) to constraint the use of model constructs. In this paper we investigated a third strategy that is based on the definition of metamodels for ADLs and their mapping to UML using model transformations.

Five strategies for representing architectural structure in UML are described by Garlan et al. [14]. They conclude that there is no single best way to do this. Furthermore, they identify a trade-off between completeness and legibility: strategies that assign different UML model elements for each ADL construct (completeness) tend to be very verbose and hence poorly readable (legibility). One of their recommendations to solve this, is to continue to use ADLs but to provide mappings to object-oriented notations. In the current paper we specified such mappings using model transformations, which makes them automated.

Where we propose to use MOF for the definition of DSMLs, Dashofy et al. [26] use XML for the definition of ADLs. It provides generic high-level XML schemas that can be extended for development of ADLs. They leverage the available tool support for XML. As we use MOF, we leverage available UML and MOF tools as well. This enables, for instance, the specification of transformations on a higher level of abstraction by a model transformation language.

8. Concluding Remarks

In this paper we proposed to combine DSML models and UML diagrams for model-driven software documentation. Where MDE approaches typically aim to use DSML models to automatically create source code, our approach complements MDE with the (partial) creation of documentation.

The main motivation for our approach is the observation that although DSMLs have clear advantages over general-purpose modelling languages, it requires considerable effort to develop graphical editors and representations. In particular, the definition and implementation of their concrete syntax or notation is much more involved than that of their abstract syntax, which is supported by technologies, such as MOF and EMF. This is a problem, as graphical representations of models are an essential part of software documentation.

Our approach uses model transformations to (automatically) map DSML models to UML models. These UML models are easily visualised as UML diagrams using available modelling tools. While the DSML models can be used for code generation and other automated software engineering tasks, these diagrams are used in the documentation. As such, our approach allows to optimise both completeness (by the ADL model) and legibility (by the UML diagram) of architecture descriptions. Furthermore, part of the documentation can be automatically updated as the software system evolves.

Application of our approach requires the definition of a DSML metamodel using MOF and mappings to UML using model transformations. This needs to be done once for each DSML used. Furthermore, a means to create associated models is required. We gave several examples for this. Compared to the development of a complete graphical editor for the defined metamodel, our approach is more lightweight.

We evaluated our approach in the domain of software architecture, for which we defined MDAV. It refines the industry standard for architecture documentation (IEEE Std 1471-2000) by linking architectural views (documentation) to architectural models using model

transformations and UML. MDAV is easily generalised to other domains. As an example, we discussed an industrial application in the domain of control systems.

Currently we are investigating how the proposed model transformations can best be integrated with existing tooling and development processes. Another problem we are investigating is the (automatic) derivation of metamodels (e.g., based on MOF) from grammars (e.g., based on EBNF). A solution to this problem increases the effectiveness of our approach when applied to existing DSMLs that are not based on MDA technology.

Acknowledgement Part of the research described in this paper was sponsored by NWO via the Jacquard Reconstructor project.

References

[1] Jean Bézivin. On the unification power of models. *Software and Systems Modelling*, 4(2):171–188, May 2005.

[2] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC '97/FSE-5: Proc. 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 60–76. Springer-Verlag, 1997.

[3] Duncan Doyle, Hans Geers, Bas Graaf, and Arie van Deursen. Migrating a domain-specific modeling language to MDA technology. In *Proc. 3rd Int'l Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ateM 2006)*, 2006.

[4] Arie van Deursen and Paul Klint. Little languages: Little maintenance? *J. Software Maintenance: Research and Practice*, 10(2):75–92, December 1998.

[5] Bas Graaf. Model-driven consistency checking of behavioural specifications. In *Proc. 4th Int'l Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, 2007. Accepted for publication.

[6] Bas Graaf, Sven Weber, and Arie van Deursen. Model-driven migration of supervisory machine control architectures. *J. Systems and Software*, 2007. Accepted for publication.

[7] OMG. Meta-object facility (MOF). `http://www.omg.org/mof`, 2005.

[8] Eclipse Foundation. Eclipse modeling framework (EMF). `http://www.eclipse.org/emf`, 2005.

[9] Christian F.J. Lange, Michel R.V. Chaudron, and Johan Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):40–46, March 2006.

[10] OMG. MDA. `http://www.omg.org/mda`, 2005.

[11] Philipe Kruchten, Henk Obbink, and Judith Stafford. The past, present, and future of software architecture. *IEEE Software*, 23(2):22–30, March 2006.

[12] Nenad Medvidovic, David S. Rosenblum, David F. Redmiles, and Jason E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Trans. Softw. Eng. Methodol.*, 11(1):2–57, 2002.

[13] Bas Graaf, Marco Lormans, and Hans Toetenel. Embedded software engineering: The state of the practice. *IEEE Software*, 20(6):61–69, November–December 2003.

[14] David Garlan, Shang-Wen Cheng, and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. *Science of Computer Programming*, 44:23–49, 2002.

[15] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures:Views and Beyond*. Addison-Wesley, 2002.

[16] IEEE-1471. IEEE recommended practice for architectural description of software intensive systems. IEEE Std 1471–2000, 2000.

[17] Object Management Group. XML metadata interchange formal specification. `http://www.omg.org/technology/documents/formal/xmi.htm`, January 2002–03.

[18] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In *Proc. Model Transformations in Practice Workshop at MoDELS2005*, 2005.

[19] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Software Engineering and Methodology (TOSEM)*, 6(3):213–249, July 1997.

[20] David Garlan, Robert T. Monroe, and David Wile. ACME: architectural description of component-based systems. In *Foundations of component-based systems*, pages 47–67. Cambridge University Press, 2000.

[21] Timothy J. Grose, Gary C. Doney, and PhD. Stephan A.Brodsky. *Mastering XMI. Java programming with XMI, XML, and UML*. OMG Press. Wiley, 2002.

[22] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces: an initial appraisal. In *Confederated Int'l Conferences CoopIS, DOA, and ODBASE 2002*. Springer-Verlag, 2002. Industrial Track.

[23] N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. PhD thesis, Technische Univ. Eindhoven, 2004.

[24] Eclipse Foundation. Eclipse graphical modeling framework (GMF). `http://www.eclipse.org/gmf`, 2006.

[25] Frédéric Fondement and Thomas Baar. Making metamodels aware of concrete syntax. In *Proc. 1st European Conf. Model Driven Architecture - Foundations and Applications (ECMDA-FA 2005)*, volume 3748 of *Lecture Notes in Computer Science*, pages 190–204. Springer-Verlag, 2005.

[26] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Software Engineering and Methodology*, 14 (2):199–245, April 2005.

TUD-SERG-2006-019a
ISSN 1872-5392