

Business Process Modeling: Defining Domain Specific Modeling Languages by Use of UML Profiles

Steen Brahe and Kasper Østerbye

IT University of Copenhagen, Copenhagen, Denmark
stbr@itu.dk, kasper@itu.dk

Abstract. General-purpose modeling languages are inadequate to model and visualize business processes precisely. An enterprise has its own vocabulary for modeling processes and its specific tasks may have attached data that define the tasks precisely. We propose using Domain Specific Modeling (DSM) languages to model business processes, such that an enterprise can define its own DSM language(s) capturing its vocabulary and data requirement. We suggest using UML profiles and UML activity diagrams as the semantic base for these DSM languages and present tools that are able to create a DSM language and tool support for a given domain. One tool, called ADSpecializer, can generate a UML profile and its tool support of a given application domain. The other tool, ADModeler, is used to create UML activity diagrams within such a domain-specific UML profile. The two tools enable an enterprise to efficiently define and utilize their own DSM language.

1 Introduction

Model Driven Engineering (MDE), as an approach for describing and implementing business processes, is believed to speed up the development time and be less error prone compared to traditional software development. Several standards have been proposed for modeling and implementing business processes [2]. Based on experiences at using MDE for describing and implementing business processes in a large Scandinavian bank, we recognize a need to have domain specific modeling languages to be able to succeed using the MDE approach. Each enterprise has its own terms for modeling business processes and has enterprise specific implementation patterns for these terms. The development of a common vocabulary in a large enterprise is crucial for efficiency. To tailor modeling tools efficiently to this common vocabulary is therefore a prerequisite for us to apply MDE.

A general language is too abstract to be used by people working in a specific domain. As Bézivin and Heckel state [1 p. 1], “model-driven approaches to software development require precise definitions and tool support for modeling languages, their syntax and semantics”.

We see at least three obstacles to use a general purpose modeling language compared to a domain specific modeling (DSM) language for business process modeling:

- *Semantics.* Specific semantics for custom tasks like RegisterInvoice cannot be defined. A modeler has to remember to define necessary data when using the task

in models and there is no tool support for providing and validating the data. A transformation engine does not recognize a task like RegisterInvoice because it is modeled as a general task.

- *Visualization.* There is no customized visual presentation of the model. Visualization is important because different people such as users, business analysts, architects and developers all have to understand the model.
- *Abstraction.* A business process may be modeled at a high abstraction level. A task such as RegisterInvoice may not have a simple implementation as e.g. a web service invocation. Instead, it could have an implementation pattern, for instance a sequence of three web service calls, and mechanisms for handling exceptions. These details are not relevant for the model, but have to be modeled when using a general language to make transformation to an implementation possible.

The primary argument against using DSM languages and customized tasks for each enterprise or even each business unit inside the enterprise is that the set of necessary languages and tasks to define will continue to evolve. We address this argument by providing tool support for definition and generation of custom tasks and new languages.

This is in line with Bézin and Heckel [1, p. 1] who state “In order to support model-driven development in a variety of contexts, we must find efficient ways of designing languages, accepting that definitions are evolving and that tools need to be delivered in a timely fashion”. Software systems are evolving all the time and enterprises will also have to extend and enrich their DSM languages. To do this efficiently they need ways to get customized modeling tools for the extended DSM languages.

We have developed two Eclipse-based UML2 tools, *ADModeler* and *ADSpecializer*. *ADModeler* is a plug-in that implements a UML activity diagram editor. *ADSpecializer* can define and generate UML profiles and data entry wizards encapsulated as Eclipse plug-ins for *ADModeler*. The modeler who uses a DSM language generated by *ADSpecializer* is not aware that she is modeling in UML. Both the language and tool support appear domain specific.

1.1 Background

The Model Driven Architecture (MDA) initiative by the Object Management Group (OMG) is an implementation of the general MDE approach for developing software around a set of standards like MOF, UML, CWM etc. [5]. UML is a visual language for specifying, constructing and documenting software systems [4]. It is a broad-spectrum language and consists of several diagram types. One of these, the activity diagram, has modeling of organizational processes as one of its purposes. UML is defined by the Meta Object Facility (MOF) [3]. MOF is a meta-meta model because it is used for defining other meta-models like UMF. MOF is defined by itself.

When using MDA standards, there are two possible approaches for creating DSM languages. The first approach is the definition of a new language based directly on MOF. Such a language becomes an alternative to UML. The Common Warehouse Meta model (CWM) is an example of such a language. The syntax and semantics of the elements of the new language can be defined to match the specific domain.

The second approach is based on specialization of the existing UML entities using UML profiles. The intention of profiles is to give a straightforward mechanism for adapting UML with constructs that are specific to a particular domain, platform, or method. A profile is constructed by using the extensibility elements: stereotypes, tagged values, and constraints. Stereotypes are specific meta-classes, tagged values are standard meta-attributes, and constraints are restrictions on how an element can be used in models. Using profiles is considered a lightweight method of defining a DSM language, while basing the language on MOF is considered a heavyweight method.

UML Profiles have been made for many specific purposes. For example several profiles have been defined for business process modeling [13, 18] or for implementation technologies such as J2EE, where refined UML Class diagram differentiate between home and remote interfaces. Each of these profiles defines UML for a particular context.

Meta-modeling tools like MetaEdit+ [12] and GME [7] show that it is possible to provide generic tool support for domain specific modeling languages. At present such tools do not exist for MOF although work is going on in projects like GMF (Graphical Modeling Framework)[22]. In contrast, the use of UML profiles for customizing the modeling language is supported by several UML modeling tools.

Business process models are sufficiently similar to the fundamental abstractions of activity diagrams so that we believe using profiles for defining DSM languages is feasible. UML Activity diagrams can model most of the workflow patterns described in [9] and have more expressive power than most of the industrial workflow management standards [10, 11] for implementing business processes. It is therefore a natural choice to use activity diagrams for modeling business processes.

1.2 Our Work

We use UML activity diagrams and UML profiles to create domain specific modeling languages for business processes. Activity diagrams have the formal expressive power to formulate the business processes we want to model. UML is a specification and is supported by general tools such as Rational and Poseidon, which support creation and use of profiles.

However, using profiles for domain specific modeling in general modeling tools requires good knowledge of both UML and profiles as the general tools do not support modeling directly in domain specific terms. The usability of the tools remains low, in particular:

1. The abstract notion of actions lies far from concrete tasks like “change reservation”. This makes the tools less useful to domain experts.
2. There is no way to customize how attributes for a particular stereotype such as “RegisterInvoice” should be entered.
3. There is no design-time validation of attribute values or model element relationships.

The general tools do not support these requirements, and the commercial tools are not sufficiently open to tailor them. We will therefore work with the open source tool Eclipse [19]. The UML2 eclipse project [20] provides an implementation of the

UML2 specification and is based on the Eclipse Modeling Framework (EMF) [21] which implements a subset of MOF.

We address the vision of providing enterprise specific process modeling tools in a two-step fashion. First, our ADModeler is a general-purpose extensible and open source UML activity-diagram editor, and is to our knowledge the first such for the Eclipse framework. Special emphasis has been placed on rendering UML profiles containing specification of icons for each stereotype, and the definition and management of mandatory auxiliary data.

Secondly, our ADSpecializer enables efficient development of enterprise specific profiles. It can generate a profile for use by ADModeler. It creates icons, images and text to present the specific profile in ADModeler, and wizards to enter data for the specific tasks. The Eclipse framework provides a rapid and seamless profile-development cycle for testing plug-ins, which we leverage by making ADSpecializer generate the profile as an Eclipse plug-in. ADSpecializer is a no-coding-required tool and requires only limited knowledge of UML activity diagrams.

We define two different roles, a tool developer and a modeler. The tool developer is a person responsible for developing tools in an enterprise. He uses ADSpecializer to create DSM languages. The modeler is a domain expert. She uses ADModeler with extensions created by the tool developer to model business processes precisely in domain specific terms.

The usability of ADModeler is enabled for a particular domain as the specific tasks are available directly from the editor's tool palette, addressing point one above. When adding a task, the modeler is presented a wizard to define data for the attributes of the task. This addresses point two above. Point three is addressed by allowing a tool developer to define validation rules in the generated wizards for the different tasks, so consistency in the model is ensured.

A tool developer can use ADSpecializer to create a DSM language and customized tool support for it with only limited insight into UML. Further, using the ADModeler it is possible for a modeler to work with domain specific terms without any knowledge of UML.

The rest of the paper is structured as follows: In section 2, we give an example of using our tools to model processes in a human family. We first identify domain specific tasks for modeling processes in the family, then we create a new language for modeling processes using the ADSpecializer, and last we create a model of the process of getting home from work using the newly generated DSM language. In section 3, we describe the architecture of the tools, and in section 4 and 5 we describe related work, give a summary, and outline future work.

2 Example: DSM Language for Processes in a Family







We illustrate the power of defining a DSM language and a customized tool for a particular domain by looking at the processes in a human family. The family domain has been chosen since it is well known to all and easy to illustrate. Example of processes in a family are *Getting home from work*, *Go to the cinema* and *Drive on vacation*.

First, we define the language. We ask: what specialized tasks do we require to model processes in the family, what are the attributes for these tasks, and what new data types do we need. Secondly, we use the ADSpecializer to define the language and to generate a plug-in to the ADModeler. Thirdly, we use the generated plug-in together with the ADModeler to model the process of getting home from work.

2.1 Language Definition

We limit the language for modeling processes in the family to deal with six different task types. These are Transport, Clean, Cook, Shop, Relax and Nurse Kid, and are described below in table 1 including the images used for their graphical representation.

Table 1. Custom tasks for the Family DSML

Task	Icon	Description
Transport		Transport family members to a destination using some kind of transportation, e.g. a car, a bus or a train
Clean		Clean a room. The cleaning can be of different types, e.g. vacuum cleaning, wash the floor etc.
Cook		Cook a meal. It must be specified which kind of meal should be created; breakfast, lunch or dinner
Shop		Do some specific shopping, such as groceries or clothes.
Relax		Take some time for watching TV, exercise or sleep. For the task it must also be specified for how long time relaxation can be done.
Nurse kid		Take care of the children, play with them, put them to bed, etc.

To be able to define these tasks and their attributes we must also define some data types. For example we must have a data type defining that we can choose between the kitchen, the toilet and the living room when we use the Clean task and have to decide which room to clean. Table 2 lists the different data types for our new language. Here we define only Enumeration data types, although we could also have defined composite data types containing attributes of other data types. When we have specified the required data types, we can define the custom tasks and their attributes. These can be found in table 3.

Table 2. Data types for family DSML

Data type	Possible values
TransportationType	Car, Bicycle, Train, Bus
CleanType	Vacuum clean, Wash floor
RoomType	Kitchen, Toilet, Living room
MealType	Breakfast, Lunch, Dinner
ShoppingType	Grocery, Clothes, Lumberyard
ActivityType	Sleep, Play soccer, Watch TV
NurseType	Play, Bath, Change nappies, Put to bed

Now, after having described the custom tasks, their attributes, and the required data types, we can generate the language using the ADSpecializer.

Table 3. The custom tasks and their attributes

Task	Attributes	Type	Description
Transport	meansOfTransport destination	TransportationType String	Which transport? Where to go?
Clean	room cleanWhat	RoomType CleanType	What room to clean? What to clean?
Cook	Meal Persons	MealType Integer	Which meal to cook? Number of persons.
Shop	shopKind	ShopType	What to shop?
Relax	activity duration	ActivityType integer	What to do? How many minutes?
Nurse kid	activity duration	NurseType integer	What to do? How many minutes?

2.2 Language Creation

The ADSpecializer creates an extension to the ADModeler after a tool developer has used a wizard to define the previously described language. The wizard contains three steps. First, the language or the profile is named and described. Then the tasks are defined, and at last, the custom data types and attributes for the tasks are defined.

Completing the wizard, a new Eclipse plug-in project is created containing an UML profile with stereotypes, attributes and data types as defined in the wizard. Further, the plug-in extends the ADModeler so the defined tasks can be used within ADModeler. The generated plug-in project also contains generated wizards for each task to be used to collect data for the defined attributes when a modeler inserts a task of a given type into a model.

2.3 The Process of Getting Home from Work

While it would have been useful to demonstrate a process from an industrial application, we have chosen to show a process from the domain of a human family because it

is well known to all. Here we describe a simplified process of getting home from work and try to model it by using our domain specific language.

After getting off from work, you drive to the daycare to pick up your child. Then you go to the grocery shop to buy food for dinner and, then you drive home. At home a lot of things now happen in parallel; you start cooking dinner, you have to check the nappies on your kid and optionally change it, also you have to play with the kid, and you have to clean the floor. When dinner is ready, you stop cleaning, and the family eats. After dinner, you put your kid to bed, and exhausted, go to relax in front of the television for an hour before going to sleep.

This process has been modeled in ADModeler using the Family language and can be found in figure 1, which also illustrates the ADModeler working with the generated plug-in containing the Family language. In the tool palette to the right, all the customized tasks as defined in table 1 can be found. A task instance can be dragged from the palette onto the model. As the figure illustrates, we have also customized the general UML decision and merge nodes to use a question mark as image. Doing this makes the tool more intuitive to use by a domain expert.

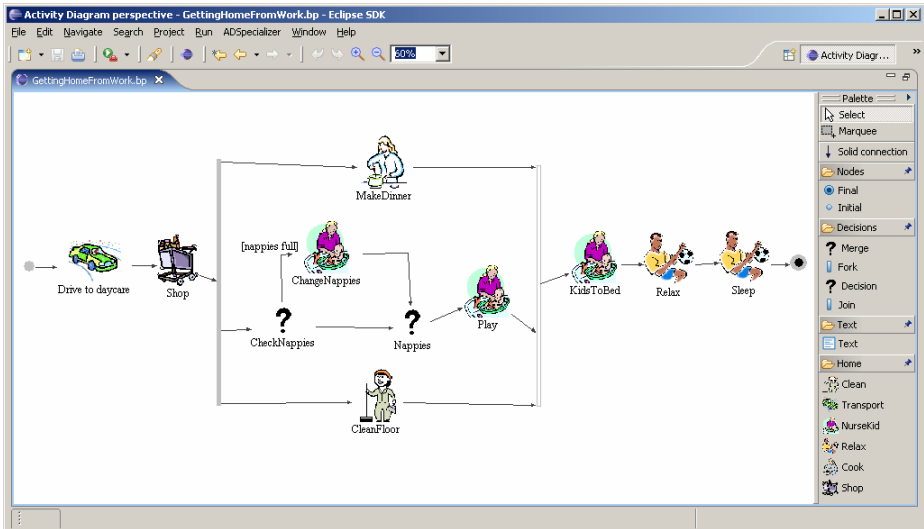


Fig. 1. ADModeler with the Family DSM language extension and modeling the *Getting home from work* process

Still the modeler could be customized further, e.g. unnecessary menus and toolbars could be removed from the tool, and a special view for accessing attribute data could be created.

Whereas the customized diagram is syntactic sugar over plain UML, the semantics of the task instances is the real force of our approach. When a task instance is added to the model, the modeler is presented with a customized wizard for collecting data

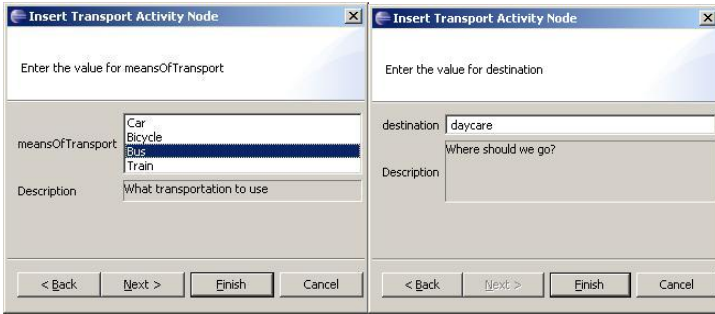


Fig. 2. Generated wizard pages for defining data for the Transport Task attributes

for the attributes defined for the task. Figure 2 shows the two generated wizard pages for entering attribute data for a Transport task, which are the transportation type and the destination. A tool developer can customize these pages if the generated ones are insufficient for a particular task, e.g. if some specific validation is required or data has to be retrieved from a database.

The example illustrates having a DSM language when modeling and having tool support for this DSM Language. We gain a more intuitive model, precise semantics and guided definition of required data. Our tools have made the process of creating DSM languages and tool support for them automatic with no need for technical insight into UML and eclipse plug-in development. The example shows that using activity diagrams and profiles for creating DSM languages using our tools is straight forward. Using the generated tools hides the complexity and generality of UML and instead provides domain specific terms, symbols and wizards to be used directly by the modeler.

3 Tool Details

In this section, we give an overview of the ADModeler and ADSpecializer tools, how they use meta-models, how ADModeler can be extended, and how ADSpecializer automates the task of creating such extensions.

3.1 ADModeler

The ADModeler is a general-purpose UML activity diagram editor but provides an Eclipse extension point that enables tool developers to extend the editor for specific purposes, i.e. they can define their own domain specific languages and customize the editor and tool palette. ADModeler will appear as if it was created for the specific domain. A model can be defined by adding instances of the domain specific tasks directly from the palette. A domain specific task represents a specific UML ActivityNode, for instance an Action or a DecisionNode with an applied stereotype such as Transport which indicates an action of transporting oneself from one destination to another. The stereotype is defined in a profile that is contained in the plug-in that extends ADModeler.

Furthermore, the tool developer is able to define how a modeler is supported in providing attribute data for the specific tasks. This is done by creating a wizard containing a number of wizard pages for each custom task. The wizard is able to validate input from a modeler before an element is inserted into the model. The validation check can be everything ranging from simple validation of text strings to validation against values in databases or from web services. Wizards are not always considered a good strategy for providing tool support [23, p. 126] so this approach may be revised in the future.

ADModeler provides a graphical editor for creating and editing UML2 activity diagrams. We have built the editor using a number of open source eclipse plug-ins providing a framework for making graphical editors and implementations of MOF and UML 2.0 specifications. These plug-ins are

- Graphical Editor Framework (GEF). The project provides an easy way to create a rich graphical environment based on a model.
- Eclipse Modeling Framework (EMF). The EMF project provides an implementation of a subset of the MOF specification. Using this project enables a tool developer to define his own modeling languages based on MOF.
- UML2. The project provides an implementation of the UML2 specification and builds on EMF. The project makes it possible to create models which conform to the UML2 specification although it does not provide any graphical annotations or possibilities of making visual diagrams of models.

3.1.1 Meta-models in ADModeler

Because the Eclipse UML2 project contains no implementation of the UML2 Diagram Interchange Specification or other visual data, we have to decide how to define visual information for an activity diagram. We could define a profile containing the visual information and apply it to all model elements. But a lot of irrelevant information would pollute the model. Another approach could be to create a new meta-model which contains both visual and semantic information and from which UML could be exported. We have chosen neither of these. Instead we have created a new MOF based meta-model called ADModel representing all visual information about the activity diagram. This meta-model does not contain any semantics. Instead it wraps or links to the UML2 meta-model, which represents the semantic model of activity diagrams. The ADModel meta-model could be thought of as a decorator of the UML2 meta-model.

When creating a model in ADModeler two models are produced. One model based on the ADModel meta-model contains all visual information and one model based on the UML2 meta-model contains all semantic information. The strengths of this approach are:

- Separation of visual and semantic information in two models.
- Semantic model is directly available from file system for other UML tools like modeling tools or transformation engines, which do not require visual information.
- Simple visual model extensible for plug-ins.

Because the UML model is not encapsulated in another model, no extraction or export has to be done from the visual model. The UML model can be edited directly,

except actions like adding or deleting elements, in another tool and the corrections will be reflected in the editor when shown in ADModeler.

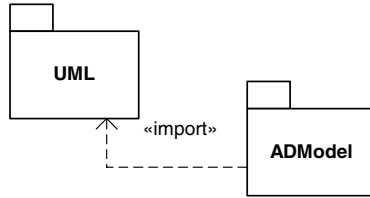


Fig. 3. Meta-model dependency from the UML meta-model

The meta-model used by ADModeler is illustrated in figure 3 and figure 4. Each element in the meta-model has a reference to an element in the UML meta-model. The most interesting part of the meta-model is the Node element which represents the ActivityNodes, or the building blocks, in the activity diagram.

It contains attributes for various visual presentations like coordinates and size. It further contains a typeId attribute and has a link to the abstract UML class ActivityNode. Concrete implementations of the ActivityNode class include classes like Action, Decision-, Join-, Fork-, and Merge nodes. An instance of a Node in a concrete model will have a reference to an instance of one of these concrete ActivityNode types.

The typeId attribute at the node indicates which kind of ActivityNode and optional stereotype the Node represents. Using a typeId and a reference to the abstract ActivityNode enables us to make the model extensible for others. For example, the Transport task contained in the Family language has a typeId equal Family.Transport and extends an Action node. It also represents the stereotype Transport. When a Transport task is inserted into a model, a Node and an Action instance is created. The Transport

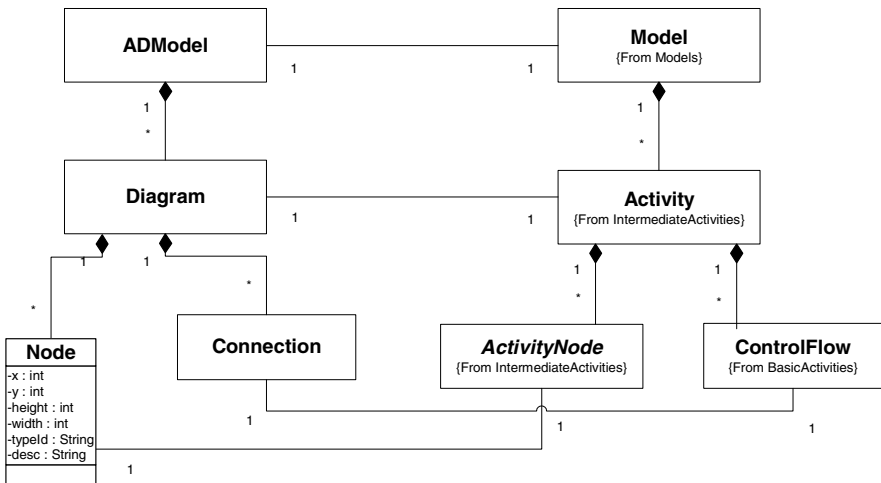


Fig. 4. The ADModel meta-model and references to elements in the UML meta-model

stereotype is applied to the action. The Node instance has a link to the Action instance and a typeId equal Family.Transport.

The tool provides standard typeId's for the most common ActivityNodes; Initial-Node, ActivityFinalNode, Action, DecisionNode, MergeNode, ForkNode and Join-Node. Next section provides more information about how to define an extension to the ADModeler.

3.1.2 Extension Point

ADModeler provides an extension point for extending the editor and its underlying meta-model. By default, the modeler supports modeling with seven different node types as described above. These are registered in a NodeRegistry which maps a typeId to a specific kind of UML ActivityNode, an optional stereotype, an icon, image, label, description and group, and a wizard for collecting data for stereotype attributes. When opening the editor, its tool palette is built by reading the NodeRegistry and creating a tool for each entry.

Each element in the palette contains a typeId. When an element is dragged onto the editor, ADModeler from the NodeRegistry retrieves the kind of ActivityNode to instantiate, the stereotype to apply at the ActivityNode, a wizard, etc. based on the typeId. After looking up the typeId it presents the wizard to the modeler to collect data. Then it instantiates the concrete ActivityNode type, optionally applies the stereotype, sets stereotype attributes and at last presents the Node in the diagram using the image registered in the NodeRegistry.

To extend ADModeler, one has to provide the data described in table 4.

Table 4. ADModeler extension point attributes

Attribute	Description
PaletteLabel	The label to be used in the tool palette, e.g. <i>Transport</i> .
PaletteTip	The tool tip text for the palette, e.g. <i>Transportation to somewhere</i> .
Group (Optional)	The tool group in which the extension should be present.
PaletteIcon path	A relative path to the icon for the palette.
EditorImage path	A relative path to the image for the editor.
ActivityNode type	The type of UML activity node, e.g. Action.
Profile path	A relative path to the profile containing the required stereotype.
Stereotype name	The name of the stereotype to be applied to the ActivityNode
Wizard class name	A wizard class for collecting data for the stereotype attributes.
typeId	A unique Id for this type to be used in the NodeRegistry, e.g. org.mda4bpm.homeprofile.Transport.

One limitation of the tool is that only the control flow part of activity diagrams can be modeled. Modeling of the object flow is not implemented. Furthermore, it does not support defining restrictions in e.g. OCL or Java for how new tasks may be used in

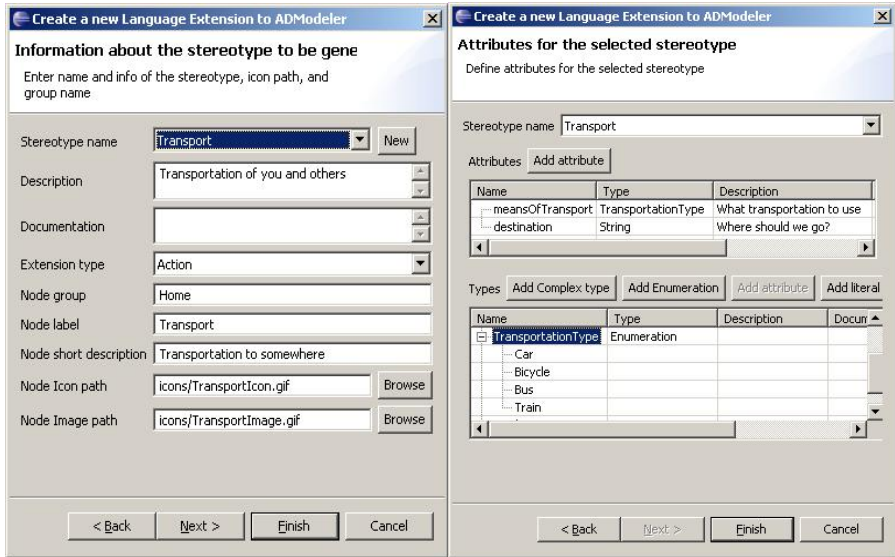


Fig. 5. Wizard pages for ADSpecializer

the model or how to validate stereotype attributes. Attribute validation can be done in the wizard class, but one has to do this in plain Java code.

3.2 ADSpecializer

To extend ADModeler, a tool developer has to create a new plug-in project and define the extension. As part of defining the extension, he has to create a wizard and an UML profile. This requires good technical insight into both the Eclipse platform and into UML. Further, it requires a UML tool supporting profiles to be able to define the profile. To aid in this task we have developed the ADSpecializer tool.

To define a new DSM language, a tool developer is guided through a wizard. Figure 5 shows the pages used to define a new stereotype. The first page defines the graphical appearance, and which UML-type that is extended. The second page is used to define the custom attributes to be associated with this new stereotype. Currently attributes of type integer, Boolean, and string, and user defined enumerations are supported. In addition, it is possible to define aggregations of such values, which we call complex types.

Complex types as well as enumerations are defined in the right hand window shown in figure 5. An additional page (not shown) is used to define the name of the profile. The data model behind the wizards conforms to a MOF based meta-model that we call ADProfile, which is shown in figure 6. In particular, complex types and enumerations are represented in the underlying model. Based on this model, ADSpecialiser generates an eclipse plug-in that contains one extension to ADModeler for each custom task defined. Further, it generates all resources required for the extension point defined by ADModeler.

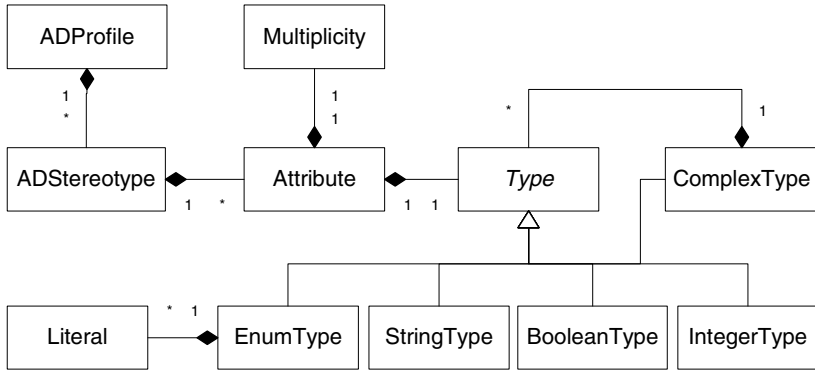


Fig. 6. Meta-model used by ADSpecializer

In the next section, we describe related work in the area of business process modeling notations and particular the use of activity diagrams.

4 Related Work

Many standards have been proposed for modeling business processes and their implementations. Two notations are dominating the modeling field. The Business Process Modeling Notation (BPMN) [15] is a graphical notation intended for business analysts. The UML activity diagrams, on the other hand, are part of the UML suite of technical diagramming notations. Both notations are able to model most of the workflow patterns described in [9] which means they are feasible for modeling business processes [10, 14]. On the implementation side, the most important standard is the Business Process Execution Language for Web services (BPEL4WS or just BPEL) [6].

Transformation rules have been proposed for both the BPMN notation [15] and UML activity diagrams [8, 13] to BPEL, so an implementation can be generated directly from a business process model.

Several others before us have used UML activity diagrams for business process modeling. Heckel and Voigt [8] suggest using a profile for UML activity diagrams for modeling business processes with the purpose of generating BPEL code. Combined with graph transformation as a meta-language for defining model transformations such models are transformed into BPEL. Heckel also presents techniques to analyze the models. Staikopoulos and Bordbar [16] have studied how the UML meta-model and the Web-services meta-models can be integrated so transformations can be facilitated. They present a method to support meta-model integration and interoperability and exemplify this with the BPEL meta-model. In [17] the same authors have used activity diagrams to capture the behavioral aspects of composing web-services and to transform these diagrams into BPEL. Eriksson and Penker have written a complete book about using UML for business modeling and have among other thing defined a profile to be used for business process modeling [18].

Common to the above-mentioned work on using UML activity diagrams and profiles for business process modeling is that they suggest using *one* profile for process modeling regardless of application domain. Our contribution is to enable enterprise specific tailoring of the modeling tools, and to give tool support for the tailoring process. We believe this tailoring is necessary to ensure the semantics, visualization, and abstraction of business process modeling as mentioned in the introduction.

5 Summary and Future Work

We have suggested UML activity diagrams as a general-purpose business process modeling language and using UML profiles for creating DSM languages for a specific enterprise.

We presented the general-purpose UML activity-diagram modeling tool ADModeler, and the ADSpecializer that automates the process of defining DSM languages and create customized tool support for them. The effectiveness and efficiency of these tools to model a solution in domain specific terms were demonstrated in the human family domain.

Several open issues remain. Currently, presence of mandatory attribute data is validated. However, we lack mechanisms to define restrictions on their values. In addition, it should be possible to constrain the manner in which concrete task types are combined (e.g. invalidate concurrent cleaning and transport by the same person). The modeling tool should be able to interpret these constraints and guide the modeler. Further, it should be possible to model object flows and to extend already defined languages with new even more specialized languages, i.e. specialize profiles.

A motivation for this work has been a wish to combine domain specific modeling with model transformations toward an implementation. For each custom task type defined in a profile, we need to define custom transformation rules and model templates representing patterns at lower abstraction levels.

In the future, we expect to evaluate the strength and weaknesses of the proposed tools for modeling business processes. We will evaluate it using real business processes together with our industrial partner. Further, we will start to work on customized model transformations and the use of model templates to automate the development of implementation specific code like BPEL.

We believe that having the combination of domain specific modeling languages, customized model transformations, model templates, and tool support for these for a single enterprise will be a crucial step towards the MDE vision: To heighten the abstraction level in software development.

References

1. Bézivin, J., Heckel, R.: Language Engineering for Model-driven Software Development. Dagstuhl Seminar Proceedings 04101(2005) 1-8
2. Havey, M: Essential Business Process Modeling. O'Reilly Media, Inc. (2005)
3. OMG: Meta Object Facility 2.0 Specification. Document id: ptc/04-10-15 (2003)
4. OMG: UML 2.0 Superstructure Specification. Document id : formal/05-07-04 (2005)

5. Kleppe, A., Warmer, J., Bast, W.: *MDA Explained: The Model Driven Architecture- Practice and Promise*. Addison-Wesley (2003)
6. BPEL: BEA, Microsoft, IBM, SAP, Siebel, *Business Process Execution Language for Web Services*, Version 1.1. (2003)
7. GME, Generic Modeling Environment, <http://www.isis.vanderbilt.edu/Projects/gme>, last accessed 29 Jan 2006
8. Heckel, R., Voigt, H.: *Model-Based Development of Executable Business Processes for Web Services*. *Lecture Notes in Computer Science*, Vol. 3098. Springer-Verlag (2003) 559-584.
9. van der Aalst, W.M.P., Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: *Workflow Patterns*. *Distributed and Parallel Databases*, 14(1) (2003) 5-51
10. Wohed, P, van der Aalst, W.M.P., Dumas, M., Hofstede, A.H.M., Russell, N.: *Pattern-based Analysis of UML Activity Diagrams*. Technical report #129, Beta Research School, Eindhoven University of Technology, December 2004
11. Dumas, M., Hofstede, A.H.M: *UML Activity Diagrams as a Workflow Specification Language*. *Lecture Notes in Computer Science*, Vol. 2185, Springer-Verlag (2001) 76-90
12. MetaEdit+, MetaCase modeling tool, <http://www.metacase.com>. last accessed 29 Jan 2006
13. Gardner, T.: *UML Modelling of Automated Business Processes with a Mapping to BPEL4WS*. Presented at 17th European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany (2003)
14. White, S.: *Process Modeling Notations and Workflow Patterns*. In L. Fischer, editor, *WorkflowHandbook 2004*. Future Strategies Inc., Lighthouse Point, FL, USA (2004) 265-294
15. White, S.: *Business Process Modeling Notation, Version 1.0* <http://www.bpmn.org/Documents/BPMN%20V1-0%20May%203%202004.pdf> May 2004. Last accessed 29 Jan. 2006.
16. Staikopoulos, A., Bordbar, B.: *A Comparative Study of Meta-model Integration and Interoperability in UML and Web Services*. *Lecture Notes in Computer Science*, Vol. 3748, Springer-Verlag (2005) 145-159.
17. Bordbar, B. Staikopoulos, A.: *On behavioural Model Transformation in Web Services*. *Proc. Conceptual Modelling for Advanced Application Domain (eCOMO)*, Shanghai, China (2004) 667-678
18. Eriksson, H.E., Penker, M.: *Business Modeling with UML*. *Business Patterns at Work*. John Wiley & Sons, Inc. (2000)
19. Eclipse Project, <http://www.eclipse.org/>
20. Eclipse UML2 project, <http://www.eclipse.org/uml2/>
21. Eclipse EMF project, <http://www.eclipse.org/emf/>
22. Eclipse GMF project, <http://www.eclipse.org/gmf/>
23. Lauesen, S.: *User Interface Design: A Software Engineering Perspective*. Addison Wesley (2005)