Aspect-oriented Concrete Syntax Definition for Deep Modeling Languages

Colin Atkinson[1] and Ralph Gerbig[1]

University of Mannheim
{atkinson, gerbig}@informatik.uni-mannheim.de

Abstract. Multi-level modeling tools provide inherent support for modeling domain scenarios with multiple classification levels. However, as the success of domain-specific modeling tools illustrates users increasingly expect to be able to visualize models using domain-specific languages. It is relatively straightforward to support this using traditional "two-level" modeling technologies, but many of the benefits of multi-level modeling would be lost. For example, in a multi-level context it is not only desirable to define concrete syntax that is applicable over more than just one instantiation level, it should also be possible to customize the visualization of model elements as they become more specialized over instantiation and inheritance levels. In this paper we present an approach for multi-level concrete syntax definition which addresses this need by using aspect-oriented principles to parametrize the visualization associated with model elements. We also explain how this is implemented in the Melanee deep modeling tool.

Keywords: aspect-orientation; deep modeling; concrete syntax

1 Introduction

Since concrete syntax definition is one of the core foundations of domain-specific, model-driven development a large number of tools supporting this capability are available today. Important examples include MetaEdit+ [17] and the Graphical Modeling Framework [11] for graphical languages, XText [9] and Spoofax [12] for textual languages and EMF Forms [8] for form-based languages. Other concrete syntax formats can also be supported such as table-based and wiki-based languages. However, these tools are all based on traditional "two-level" modeling technology which only supports two classification levels. As a result, they have no inherent capability to support the definition of concrete syntax that can "span" (i.e. automatically be applied over) multiple classification levels. In other words, they can only inherently support the application of a concrete syntax definition to instances at the level immediately below. To apply a concrete syntax to two or more levels below its definition, complex transformation and editor generation/deployment steps are needed. This not only complicates the initial defintion and use of concrete syntaxes, it significantly increase the effort involved in maintaining and evolving them.

Deep modeling has the potential to address this problem because it inherently supports multiple classification levels. However, some challenges need to be addressed to support effective level-spanning concrete syntax definition and application within a deep modeling environment. The most significant is to find a suitable tradeoff between the need to define common syntax elements that are suitable across multiple levels whilst providing the flexibility to customize the common syntax elements as and where needed. In other words, the most significant challenge is to allow the concrete syntax used to visualize model elements to reflect the specialization that inherently takes place in the instantiation and inheritance hierarchies in a deep model.

The Melanee [3] deep modeling environment addresses this problem through an aspect-oriented approach which essentially allows concrete syntax definitions to be parametrized. Using this capability it is possible for the concrete syntax applied to a model element to be customized to reflect the specialization that naturally takes place in a deep model, significantly reducing the complexity involved in defining, maintaining, and evolving different concrete syntaxes. The implemented mechanism paves the way towards general concrete-syntax repositories which can be configured for particular domains and application scenarios in a simple and straightforward way.

The remainder of this paper is structured as follows: In the next section (Section 2) the underlying deep modeling approach is introduced. Section 3 then introduces the concept of aspect-oriented concrete syntax definition while Section 4 presents pragmatic observations made during the use of the approach in three different domains. The paper closes with conclusions (Section 5).

2 OCA-based Deep Modeling

The most widely implemented deep modeling architecture is the orthogonal classification architecture shown schematically in Figure 1. Its most obvious difference to the traditional linear modeling infrastructure of the UML is that there are two "orthogonal" classification dimensions. One dimension, the linguistic classification dimension, is represented by the vertical stack of levels labeled L_2 to L_0, and the other dimension, the ontological classification dimension, is represented by the horizontal stack of levels labeled O_0 to O_2.

The linguistic levels essentially capture the organization of the model information from the perspective of how deep modeling is realized in a meta-modeling framework such as EMF. The top most linguistic level (L_2), contains all language constructs available in the deep modeling language, while the middle level (L_1) contains the user-modeled domain languages and is thus often referred to as the domain level. The ontological dimension, in turn, consists of "ontological" levels which are stacked horizontally within L_1. Each model element at this level is classified by exactly one model element at level L_2, indicated by vertically dashed classification lines. The lowest level, L_0, contains the real-world concepts modeled by L_1. The light bulb represents the concept of *EmployeeType* which captures the common properties of all employees. The group of stick-men with a wrench

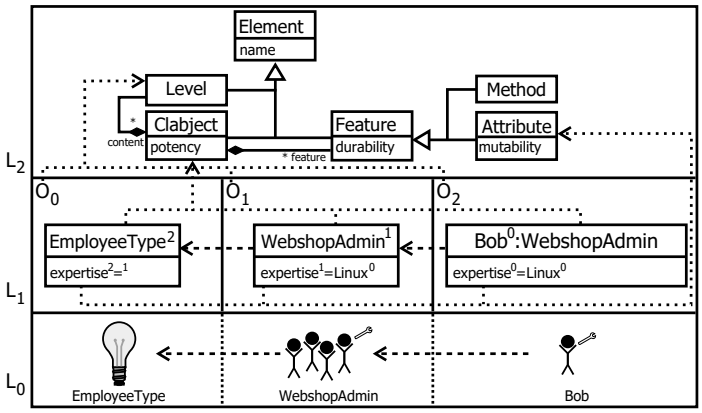

Fig. 1. The orthogonal classification architecture.

in the upper right instantiates this concept as *WebshopAdmin* (a particular type of employee) which is then further instantiated as *Bob* (a particular instance of a *WebshopAdmin*). Classes located in the middle levels (e.g., *WebshopAdmin*) are instances of the classes at higher levels and types for the classes at the lower levels. Hence, they are named "clabjects" which is a concatenation of the terms "class" and "object".

The domain level, L_1, consists of three ontological levels O_0 to O_2 which are intended to represent the domain concepts/objects. Even though three levels are shown here, the number of levels available is not limited. The concrete syntax used in Figure 1 looks similar to the UML but with some modifications to make it level-agnostic. Ontological classification is shown using horizontal dashed lines or the UML colon notation as used in object specifications (e.g. *Bob*). The most important difference to the UML is the association of numerical values with clabjects, attributes and attribute values in the form of superscripts after their names. Depending on whether they are associated with clabjects, attributes or attribute values they are respectively referred to as "potency", "durability" or "mutability". The potency specifies over how many subsequent ontological classification levels a clabject can be instantiated and thus influence the contents of a deep model. In the example *EmployeeType* has a potency of two stating that it can be instantiated at the following two levels, here *WebshopAdmin* with potency one and *Bob* with potency zero. The durability displayed next to the name of an attribute indicates over how many subsequent instantiation steps an attribute endures. In the example, *expertise* has a durability of two so all offspring of *EmployeeType* over the next two levels have to possess such an attribute, here *WebshopAdmin* and *Bob*. Finally, the mutability displayed next to the value of an attribute defines over how many levels the value of an attribute can be changed. The mutability for *expertise* of *EmployeeType* is one, hence it can be changed at the next instance level and is fixed from then on. On level O_1 of the example the *expertise* for all *WebshopAdmins* is set to *Linux* which means that instance *Bob* has to have *Linux* as his *expertise*.

3 Apsect-oriented Concrete Syntax Definition

Aspect-oriented concrete syntax definition employs concepts from aspect-oriented programing languages [14], e.g. AspectJ [13]. More specifically it allows concrete syntax definitions to be parametrized using uniquely-named join-points. Aspects can then be used to contribute to a join-point to customize the notation used to visualize a specialized version of the associated model element. An aspect contains the name of the join-point to which it is applied, a condition determining when the aspect is applied and an "advice" which is introduced to the join-point. The advice can be configured to be added before, after or to replace the join-point. Moreover, multiple aspects can be provided for one join-point.

The meta-model supporting the aspect-oriented definition of graphical concrete syntax in Melanee is shown in Figure 2. The model is unique for each type of concrete syntax format (e.g. graphical, textual, tabular etc.) to best fit the needs of that particular format. Here, for space reasons, the approach is explained only in the context of graphical concrete syntax definition, but Melanee supports the approach for other formats as well. Melanee's algorithm for visualizing model elements is designed in such a way that it can be configured to work with any concrete syntax definition model that supports the join-point and aspect concepts explained below.

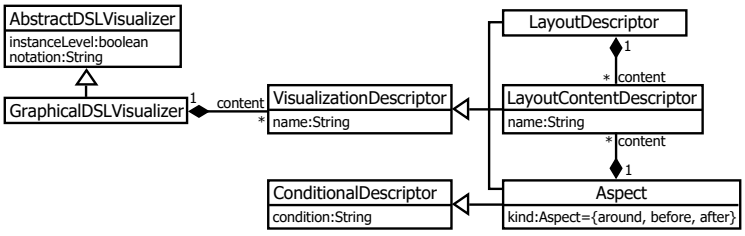

Fig. 2. The graphical concrete syntax definition meta-model.

In Melanee any model element can have multiple *AbstractDSLVisualizers* attached, defining the concrete syntax in a specific concrete syntax format. The *notation* attribute groups visualizers from the same format into "notations" so that families of symbols can be grouped into a given logical "notation". The *instanceLevel* attribute specifies whether a visualizer is applied only to the instance level or to the level at which the visualizer is defined as well. The *GraphicalDSLVisualizer* shown here defines concrete syntax in a graphical format using *VisualizationDescriptors*. This is the base class for all types of elements in a concrete syntax definition. It assigns a name to each element of a concrete syntax definition making each of those potential join-points. To explicitly declare an element as a join-point a unique name has to be assigned to it. A concrete syntax definition consist of three types of *VisualizationDescriptors*: *LayoutDescriptor*, *LayoutContentDescriptor* and *Aspect*. A *LayoutDescriptor* describes the layout which is applied to layout content (e.g. table layout, flow layout etc.). A *LayoutContentDescriptor*, which is contained by a *LayoutDescriptor*, describes the actual displayed

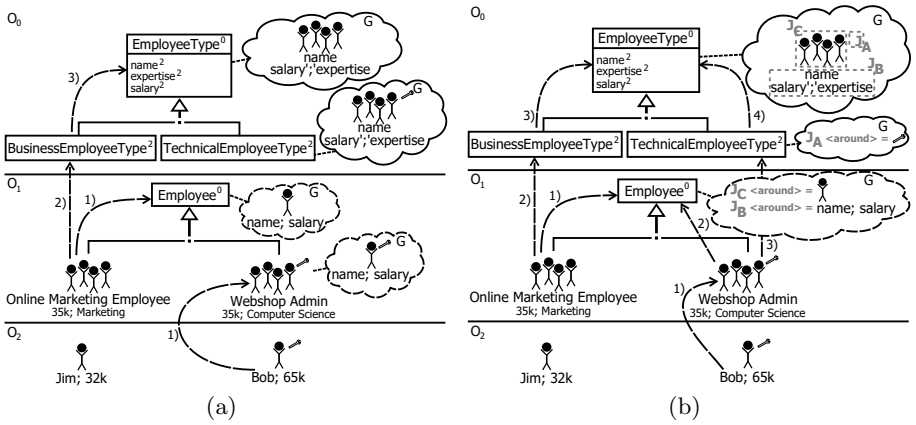

Fig. 3. Visualizer search traces: (a) not aspect-oriented, (b) aspect-oriented.

shape such as a rectangle, circle, label mappings etc. By nesting these two concepts any concrete syntax can be defined as desired. The language is designed to be similar to well known languages for describing UIs such as the Standard Widget Toolkit (SWT), and languages describing concrete syntax for diagrams such as the Graphical Modeling Framework (GMF).

Aspects for join-points are provided by the *Aspect* model element which inherits the *condition* attribute from *ConditionalDescriptors* which are executed when the *condition* holds true. *Aspects* are not only *ConditionalDescriptors* they are also subclasses of *VisualizationDescriptor* so that they can be added to *GraphicalDSLVisualizers*. The *kind* attribute defines the application strategy of the advice (*content*) of the *Aspect*. Three kinds are available – *before* (adding the content before), *after* (adding the content after) and *around* (replacing the content of the join-point). The current meta-model limits the *content* of an aspect to be *LayoutContentDescriptors*. However, this does not limit the expressive power of the model.

Based on the visualizer and aspect-oriented concrete syntax definition approach a special search algorithm is employed to generate the desired visualization for each model element in a deep model. This is based on an algorithm for two-level models [10]. It was first extended to deep modeling in [2] and since then steadily refined to the point where it supports deep aspect-oriented visualization.

Figure 3 shows an example of the "Employee" language with concrete syntax attached to model elements indicated by clouds. The concrete syntax definition uses the meta-model of Figure 2. The order in which clabjects are visited when searching for a model element visualization is indicated by dashed arrows annotated with a number. The search algorithm starts searching at the level of the clabject to visualize. First the clabject itself is visited to determine whether it has an associated visualizer, if not the clabjects in the inheritance hierarchy of the clabject are visited. If no visualizer is found at this level the types of the clabject at the level above are visited. The search continues until a suitable visualizer is found for the clabject. If none is found the pre-defined concrete syntax is used to render the clabject.

The example in Figure 3(a) shows the "aspect-unaware" search trace of the algorithm for *Bob* and *Online Marketing Employee*. *Bob* has no visualizer attached, so the algorithm visits *WebshopAdmin*, the type of *Bob*, which has a visualizer attached. The algorithm stops, returning the stick man icon with a small wrench in the upper right. To find a suitable visualizer for *Online Marketing Employee* its supertype (*Employee*) is visited which contains a visualizer. However, because its *instanceLevel* attribute is set to true (indicated by the dashed cloud) the algorithm continues searching the type level. The direct type, *BusinessEmployeeType*, has no visualizer but its supertype (*EmployeeType*) has. Hence, the search terminates and returns the group-of-stickmen icon to visualize *Online Marketing Employee*.

Figure 3(b) shows the same example but using aspect-oriented concrete syntax definition. Only one visualizer is defined at *EmployeeType* defining the join-points (dashed rectangles) J_A, J_B and J_C. Variations are modeled through aspects of kind *around* along the classification hierarchy, here at *TechnicalEmployeeType* and *Employee*. This focus on modeling variations only reduces the number of fully specified Visualizer from four in Figure 3(a) (*EmployeeType*, *BusinessEmployeeType*, *Employee*, *WebshopAdmin*) to one in Figure 3(b) (*EmployeeType*). The main difference between the two search algorithm versions is that discovered aspects are collected on the way to a model element with an aspect-less visualizer and then merged into the visualizer. The trace for *OnlineMarketingEmployee* is identical to the aspect-unaware version as no aspects are defined along its classification hierarchy. For *Bob* the search ends at *EmployeeType* and merges the collected aspects J_A, J_B and J_C into this visualizer.

Comparing the "aspect-unaware" and "aspect-aware" approaches shows that the search algorithm traverses more model elements when applying the aspect-aware variant of the algorithm. Additionally, the aspect aware version involves additional computational overhead for merging aspects into join-points, leading to a potential performance issue. The impact of this overhead has to be empirically determined by analyzing numerous practical examples. In the following section pragmatic observations about the approach are made.

4 Examples

Because of the novelty of the aspect-oriented concrete syntax definition approach only three languages using this features have been published to date. The first describes executable behavior of robots [5], the second a language for modeling executed environments in a game [1] and the third a distributed extension scenario in the domain of business process modeling and analysis [4].

The model shown in Figure 4 is part of a deep robot behavior modeling and execution framework in [5]. The excerpt focuses on the behavior modeling part of the model. At the highest level, O_0, the goal is to provide a generic language for modeling robot behavior. This is then used to create a robot behavior modeling language for a specific type of robot at O_1, here humanoid robots. One of the goals is to make the concrete syntax as configurable as possible to reflect the type of modeled behavior. For example, the *Move* action should show the coordinates alongside a small walking pictogram, while the *Gesture* action should show a

textual description of the gesture to do alongside a corresponding pictogram. To achieve this the concrete syntax definitions at the O_0 level offer join-points (J_A, J_B, J_C) which are used by the robot behavior language at O_1 for concrete syntax customization. The language is then used at level O_2 to model robot behavior. The behavior modeled in this example first instructs the robot to walk, then detect a ball and finally make an agree or disagree gesture.

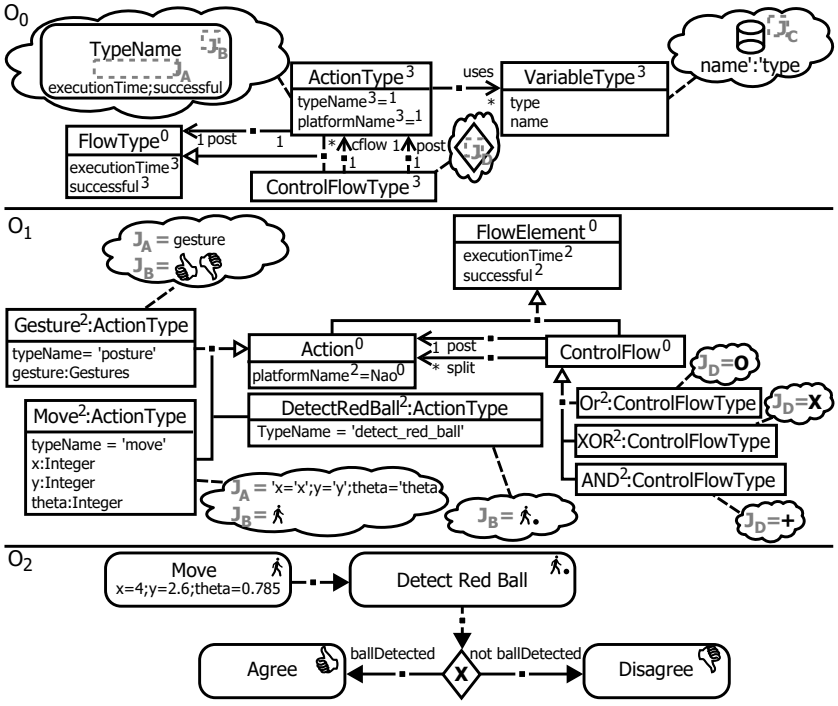

Fig. 4. Robot behavior modeling language after [5].

To realize the language without aspect-oriented concrete syntax definition features it would be necessary to define a complete visualizer for each model element at level O_1 (e.g. *Move*, *XOR*) even though the concrete syntax at level O_1 is only a slight variation of the one defined at O_0. This unnecessary replication of visualizers would significantly increase the accidental complexity involved in developing, maintaining and evolving deep models [6, 7]. A change made to the general language at level O_0 would need to be synchronized with all visualizer definitions at level O_1. Aspect-oriented concrete syntax definitions avoid this problem by making it possible to specify a generic concrete syntax definition on level O_0 that can be customized at level O_1. At level O_1 the variations are modeled only and changes to the generic syntax at level O_0 are automatically promoted to instances without any manual effort. This reduces the number of manual changes to the concrete syntax definitions that have to be made after a change and thus reduces the overall complexity involved.

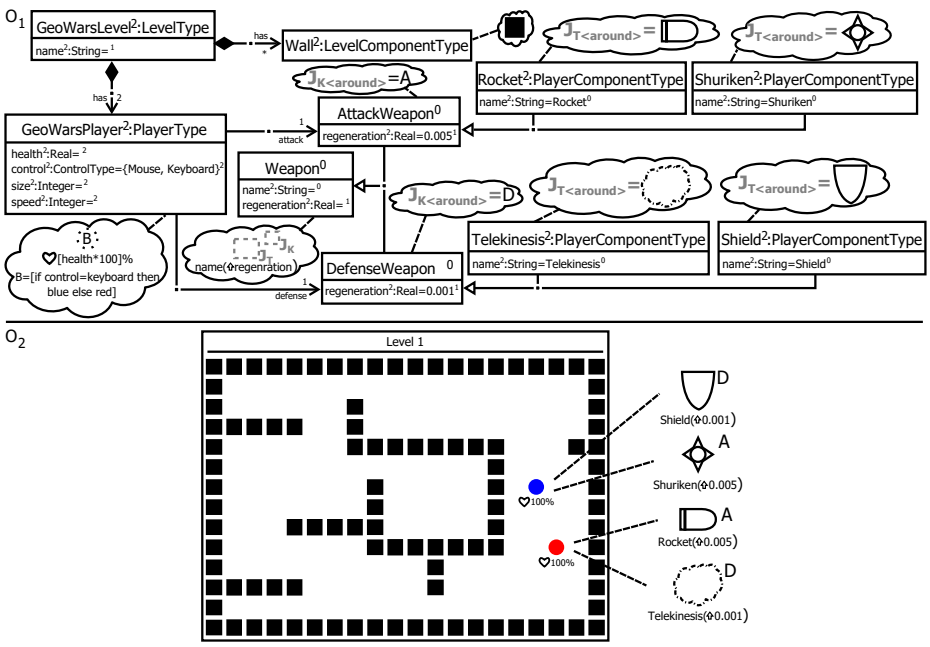

Fig. 5. Game modeling language after [1].

In contrast to the previous example where the customization takes place along the classification hierarchy, in Figure 5 aspect-orientation is used to model concrete syntax variations in an inheritance hierarchy. The example shows a model for a game environment with a focus on how the inheritance hierarchy is used to determine the concrete syntax of *Weapons* in an aspect-oriented style. At the center of each *Weapon* symbol is an icon indicating the type of weapon (e.g. *Shuriken*). The character at the top right of the symbol indicates the weapon kind (A: *AttackWeapon*; D: *DefenseWeapon*) and the text at the bottom gives the name and details of the weapon such as *regeneration* information. The only common part of all weapon visualizations is the information at the bottom. The weapon kind and type change per subclass. To optimize the concrete syntax definition, only the weapon details at the bottom are defined in the visualizer associated with *Weapon* along with two join-points — one for weapon type (J_T) and one for weapon kind (J_K). The weapon kind is associated with the *AttackWeapon* and *DefenseWeapon* subclasses since it is fixed for all of their subclasses. On the other hand, the weapon types which change for each weapon are defined locally at *Telekinesis*, *Rocket*, etc. The lowest level O_2 shows an instance of a modeled game environment which can be executed. Again the aspect-oriented concrete syntax definition reduces the number of elements involved in concrete syntax definition as only variations are modeled instead of the whole concrete syntax.

In the examples presented so far, aspect-oriented concrete syntax definition is used to reduce accidental complexity. Figure 6, however, shows a scenario that cannot be supported without this mechanism. It contains three packages created by independent parties and distributed over the internet — 1. *BPMN*:

providing a business process modeling language, 2. *MDPE*: supporting model-driven performance analysis and 3. *Security*: supporting business process security modeling.

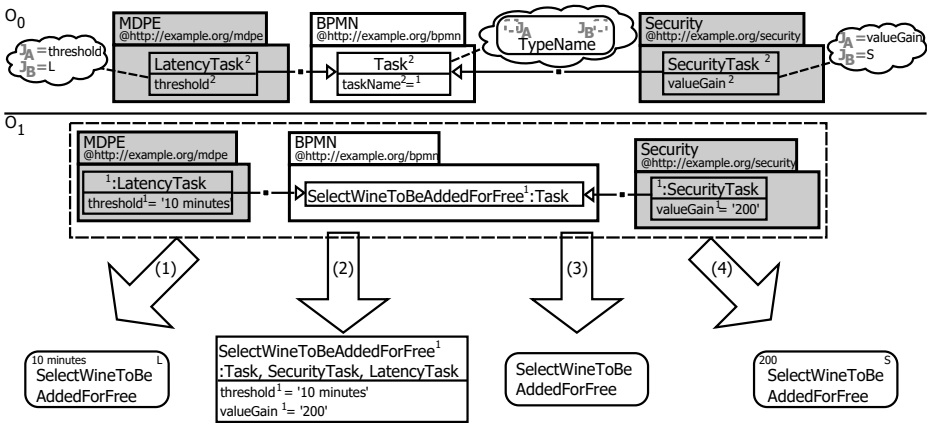

Fig. 6. Distributed business process modeling language after [4].

The *BPMN* language was defined with modeling language and concrete syntax extension in mind. To enable the latter, join-points J_A and J_B are defined in the concrete syntax definition of *Task*. These two are used by the *LatencyTask* and *SecurityTask* subclasses from the *MDPE* and *Security* packages linked to the *BPMN* package. They introduce new language constructs into the *BPMN* language and extend the concrete syntax. The rectangle at O_1 contains a detailed view of one *Task* which is an instance of *Task*, *LatencyTask* and *SecurityTask* for demonstration purposes. In a real model, however, this would be shown as one model element with multiple classifications. The four rendering options for this model instance, which it is possible to switch between on-the-fly, are shown at the bottom of Figure 6. This example shows how distributed deep models can enhance each others concrete syntax definitions in a decoupled style by using aspect-oriented concrete syntax definition. Even though the example only shows a linking depth of one, chains of linked models which contribute towards one concrete syntax can be envisaged.

5 Conclusion

Deep modeling technologies not only facilitate the definition of models that span more than one class/instance level they also allow domain-specific (i.e. user-defined) modeling languages to be used to represent them. These user-defined modeling languages often use concrete syntax definitions which span multiple classification levels, multiple inheritance levels or even multiple models. In this paper we have shown the potential of aspect-oriented concrete syntax definitions to reduce accidental complexity, positively impact model development, evolution and maintenance, and support new modeling scenarios. The three published languages and one running example described in this paper demonstrate the po-

tential advantages of the approach which is fully implemented in the Melanee tool. The authors are not aware of any other deep modeling tool supporting aspect-aware, concrete syntax definition. Besides Melanee, two other deep modeling tools explicitly support the definition of user-defined concrete syntax — MetaDepth [16], a textual tool, and DPF [15], a graphical tool. Both provide concepts for concrete syntax definition but do not support aspect-orientation.

References

1. Atkinson, C., Gerbig, R.: On the execution of deep-models. Exe 2015 – 1st International Workshop on Executable Modeling Proceedings (2015)
2. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. Software Engineering, IEEE Transactions on 35(6) (2009)
3. Atkinson, C., Gerbig, R.: Melanie: Multi-level modeling and ontology engineering environment. In: 2nd International Master Class on Model-Driven Engineering: Modeling Wizards. pp. 7:1–7:2. ACM, New York, NY, USA (2012)
4. Atkinson, C., Gerbig, R., Fritzsche, M.: A multi-level approach to modeling language extension in the enterprise systems domain. Information Systems (2015)
5. Atkinson, C., Gerbig, R., Markert, K., Zrianina, M., Egurnov, A., Kajzar, F.: Towards a deep, domain specific modeling framework for robot applications. pp. 4–15. MORSE '14, CEUR-WS.org (2014)
6. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software & Systems Modeling 7(3), 345–359 (2008), `http://dx.doi.org/10.1007/s10270-007-0061-0`
7. Brooks, F.P., J.: No silver bullet essence and accidents of software engineering. Computer 20(4), 10–19 (1987)
8. Eclipse Foundation: Emf forms. `http://eclipse.org/ecp/emfforms/index.html` (2015)
9. Efftinge, S., Völter, M.: oaw xtext: A framework for textual dsls. In: Workshop on Modeling Symposium at Eclipse Summit. vol. 32 (2006)
10. Espinazo-Pagán, J., Menárguez, M., García-Molina, J.: Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. ECMDA-FA '08, Springer-Verlag, Berlin, Heidelberg (2008)
11. Gronback, R.C.: Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit. Addison-Wesley Professional, 1 edn. (2009)
12. Kats, L.C., Visser, E.: The spoofax language workbench: Rules for declarative specification of languages and ides. OOPSLA '10, ACM, New York, NY, USA (2010)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of aspectj. In: Knudsen, J. (ed.) ECOOP 2001 (2001)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented Programming. In: Akit, M., Matsuoka, S. (eds.) ECOOP'97 (1997)
15. Lamo, Y., Wang, X., Mantz, F., Bech, O., Sandven, A., Rutle, A.: Dpf workbench: A multi-level language workbench for mde. In: Proceedings of the Estonian Academy of Sciences, 2013. vol. 62, pp. 3–15 (2013)
16. de Lara, J., Guerra, E.: Deep meta-modelling with metadepth. TOOLS'10, Springer-Verlag, Berlin, Heidelberg (2010)
17. Tolvanen, J.P., Kelly, S.: Metaedit+: Defining and using integrated domain-specific modeling languages. OOPSLA '09, ACM, New York, NY, USA (2009)