A Comparison of Configuration Techniques for Model Transformations

Dennis Wagelaar* and Ragnhild Van Der Straeten

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
{dennis.wagelaar, rvdstrae}@vub.ac.be

Abstract. MDA generally involves applying multiple model transformations. These transformations need to be applied in a particular configuration, depending on the targeted platform. Several techniques exist to manage the configuration of various software elements or components. These techniques focus on the composition rules of the various elements. A well-known application area of such techniques are Software Product Lines, in which the various features that make up a software product need to be configured. In this paper, we will investigate how several of these techniques can be applied to manage the configuration of model transformations in an MDA context.

1 Introduction

Model transformations play a central role in Model-Driven Engineering, but currently they are often applied stand-alone, much like a compiler. In an MDA context, however, various model transformations need to be combined and integrated in a build process. The type of transformations that are generally used in combination are refinement and refactoring transformations. Since these model transformations cannot always be combined safely [1], configuration techniques can help manage the composition of model transformations in an MDA context. In a model-driven build process, however, other transformations than refactoring and refinement transformations can occur (e.g. translation to other languages), which means that the configuration techniques must support model transformations in general.

Several techniques exist to manage the configuration of various software elements or components. These techniques focus on the composition rules of the various elements. The problem domain of configuration existed already within the Artificial Intelligence research area, and several techniques were used to support configuration decisions [2]. In the domain of Software Product Lines (SPL) [3], configuration techniques are used heavily to express the composition rules of the various *features* that make up a software *product*.

* The author's work is part of the CoDAMoS project, which is funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders).

A. Rensink and J. Warmer (Eds.): ECMDA-FA 2006, LNCS 4066, pp. 331–345, 2006.
© Springer-Verlag Berlin Heidelberg 2006

In this paper, we will investigate how several of these techniques can be applied to manage the configuration of refinement and refactoring transformations in an MDA context. First, an introduction of the configuration techniques is given. A running example is introduced to illustrate the model transformation configuration issues that can occur. Several criteria that are relevant for the configuration of model transformations are introduced for the purpose of comparison. After applying the selected configuration techniques to our running example, a comparison is made, based on the criteria given earlier. To conclude, we discuss the scope and impact of this comparison.

2 Configuration Techniques

This section gives an introduction of several techniques that can be used to configure model transformations.

2.1 Knowledge-Based Systems

Since the 1980's *configuration* and a *configuration task* are well-known problem domains in Artificial Intelligence (AI). A *configuration* is defined in AI as *an arrangement of parts* and a *configuration task* is defined as *a problem-solving activity that selects and arranges combinations of parts to satisfy given specifications* [2]. Knowledge-based systems are used to make configuration decisions. Other AI approaches are constraint reasoning, model-based reasoning, case-based reasoning and Description Logics.

In configuration tasks, much of the "design" work goes into defining and characterising the set of possible parts. The set of parts must be designed so they can be combined systematically and so they cover the desired range of possible functions. A specification language for a configuration task describes the requirements that configurations must satisfy. Configuration decisions are made incrementally. Depending on the alternatives chosen, different required parts will be needed, and different further requirements will be noted. This dynamic aspect of the problem suggests the use of dynamic and hierarchical constraint methods.

One of the first and best known knowledge-based configuration systems is the XCON system for configuring VAX computer systems [4]. This configuration system has a component database containing the parts and a container template database that describes how parts can physically contain other parts. The knowledge for driving the configuration task is represented mostly by production rules. The decision making process is organised in stages and subtasks.

2.2 Domain-Specific Languages

Domain-specific languages (DSLs) cover a broad spectrum of applications [5]. For the purpose of this paper, we focus on the use of DSLs for configuring refactoring and refinement transformations. The tools for developing a DSL can be classified into the following two categories:

– Grammar definition and/or meta-modelling languages and frameworks
– Transformation engines (including rule-based rewrite engines and code generators)

The main vehicle for expressing the constraints within which transformations may be combined, is the grammar definition formalism (e.g. SDF [6]) or meta-modelling language (e.g. MOF [7] or Ecore [8]). Since the expressiveness/efficiency of grammar definition and meta-modelling languages is limited, additional tools are used to express complex constraints. These tools include transformation languages (e.g. ASF+SDF [6] rewrite rules, Stratego/XT [9] and the ATLAS Transformation Language (ATL [10]) and specialised constraint languages (e.g. OCL [11]).

2.3 Feature Modelling

Feature Modelling has its origin in the Feature Oriented Domain Analysis method (FODA) [12]. Its initial goal was to document the results of analysing the commonalities and variabilities [13] of software product families/lines. Feature models allow to express variability at an abstract level without committing to any particular variability mechanism. The modelling of the semantic context of features requires some additional modelling formalisms. For example, constraints can specify the valid and invalid feature combinations. These constraints can be expressed in, for example, OCL. Several graphical tools for feature modelling exist, however, tool support for transformations and code generation is limited.

Recently some efforts have been made to formalise feature models. In [14] a translation of feature models into a context-free grammar is presented.

Feature models can also be used for configuration purposes, demonstrated by Czarnecki *et al.* [15]. In this context, a *configuration* consists of the features that were selected according to the variability constraints defined by the feature model. Configuration also refers to the process of deriving a configuration from a feature model. The configuration process is a *transformation process that takes a feature model and yields another feature model, such that the set of the configurations denoted by the latter diagram is a true subset of the configurations denoted by the former diagram* [15].

The relationship between feature modelling and domain specific languages (see subsection 2.2) is explored by van Deursen *et al.* in [16]. An important conclusion is that feature models can be translated into a DSL grammar (or meta-model). This is illustrated by their Feature Description Language (FDL), which follows the same structure as a BNF grammar.

3 Running Example: Instant Messaging Client

As a running example, we use the case study of an instant messaging client, which has been designed using the Unified Modeling Language (UML). Fig. 1 shows a UML class diagram of parts of the Platform-Independent Model (PIM)

for a simple instant messaging client. The instant messaging client is able to send and to receive messages over different kinds of networks (e.g. Jabber/Internet or SMS)[1]. It also keeps a list of contacts for each supported network. The InstantMessagingClient both uses and implements the ExceptionReporter interface: it reports raised exceptions either on the command line or on a Network that implements ExceptionReporter (e.g. a Loop-back network). The design is split up in a model, edit, view and networking part, each in their own package (the edit and view packages are not shown). Concrete view and network types are specified in separate models. These “satellite” models can be merged with the central PIM using a merging model transformation.

Fig. 1. Partial PIM class diagram for a simple instant messaging client

The example PIM contains several elements that are not available in the programming language used for the target platform. These elements are the “Applet”, “Observer”, “Observable”, “subscribe” and “Singleton” stereotypes, the “String”, “Integer”, “Exception” and “OclAny” data types, association relationships and specifications of operations (e.g. in OCL, a dynamic diagram or an Action Language). Model transformations are used to translate each of these elements to one or more elements that are available in the target programming environment. The following transformations are applied:

– There are two alternative refinement transformations that generate implementing attributes for associations. Each of them uses a different API for representing collection types.

[1] The Network class in the model actually represents a network connection; this is why InstantMessagingClient “owns” Network.

- Depending on the result of these transformations, there are two alternative transformations that generate accessor methods for each attribute, again using different collection type APIs. Note that if one chooses a particular "associations-to-attributes" transformation, one has to choose the "attributes-to-accessors" transformation that uses the same collection type API.
- There are two alternative transformations that generate observer pattern infrastructure: one uses existing API and the other generates all infrastructure. Since the "observer" transformations need to adapt accessor methods, they depend on the "attributes-to-accessors" transformations.
- There are two alternative transformations that either generate Java applet infrastructure or J2ME MIDlet infrastructure.
- There is one transformation that generates singleton pattern infrastructure. This transformation needs to take into account the initialisation control flow of a class, which is non-standard for applets and MIDlets (the singleton pattern is not allowed to create an object). Hence, this transformation depends on the applet infrastructure transformations.
- There is one transformation that generates wrappers for asynchronous methods.
- There are two alternative transformations that translate any OCL types to native Java types. One uses the simple Java collections, which are provided by all Java implementations. The other uses the Java 2 collections framework. Note that these are the two collection type APIs that are also used in the "associations-to-attributes" and "attributes-to-accessors" transformations. Hence, one again has to choose the transformation with the same collection type API as has been chosen before. In addition, this transformation should be run after all other transformations that may introduce references to OCL types.

After these transformations are run, a final transformation is run that translates the model into code (i.e. a code generator).

4 Criteria for Comparison

Model transformations have specific properties that impose their requirements on the configuration technique. This section discusses the requirements for each of these properties and states the comparison criteria for evaluating the requirements.

4.1 Generality

Model transformations are more general than the software model they are applied to; they can also be applied to models of another software product line. The configuration rules for the transformations must therefore be separated from the configuration rules of the software product line itself. This way, the configuration

rules of the transformations are reusable over multiple software product lines. The comparison criteria can be summarised as follows:

- modularity (of configuration rules);
- reusability (of configuration rule modules).

4.2 Mutual Conflicts and Dependencies

The result of applying two or more particular transformations may yield an inconsistent result or two transformations cannot even be executed in combination at all [1]. This issue is generally known as the *feature interaction problem* [17]. Such constraints are inherent to the specific transformations and the configuration rules must be sufficiently expressive to capture them. In addition, the configuration rules must express the interaction constraints in an efficient way, allowing for better maintainability of the configuration rules. To allow the configuration technique to scale up to a large number of transformations, it must be possible to check the transformation interaction constraints locally. The comparison criteria can be summarised as follows:

- expressiveness (of interaction constraints for transformations);
- maintainability (of configuration rules);
- locality (of configuration rule verification).

4.3 Platform Dependencies

The result of applying a particular refinement transformation may introduce requirements on the execution context (or target platform) of the software (e.g. require library `javax.swing` or `javax.microedition.lcdui`). It is possible that the selected transformations impose platform dependencies that cannot be satisfied by any concrete execution context (that is currently available). In [18], a method for specifying context constraints independently from concrete context is described. This method uses a query that searches for concrete context *instances* that satisfy a particular context constraint. Explicitly including this query into a configuration model results in a higher-order model. It is sufficient to enable the inclusion of the parameters for such a query, however, since the query never changes. This can be done by annotating the model transformation rules in the configuration. The comparison criteria can be summarised as follows:

- extensibility (of configuration rules with annotations).

5 Case Study

In this section, the configuration techniques introduced in section 2 are applied to the Instant Messaging running example. The particular advantages and limitations are discussed for each technique.

5.1 Knowledge-Based Systems

A knowledge-based formalism that promises useful properties for configuration is Description Logics [19]. An implementation of Description Logics that has proven its scalability and modularity is the OWL-DL ontology language [20]. Below is an ontology that describes the configuration rules for our running example[2]:

$$
\begin{aligned}
Refinement &\sqsubseteq owl:Thing \\
implies &: Refinement \times Refinement\ (transitive) \\
Java1Refinement &\sqsubseteq Refinement \\
Java2Refinement &\sqsubseteq Refinement \\
Accessors &\sqsubseteq Java1Refinement \\
Java2Accessors &\sqsubseteq Java2Refinement \\
Applet &\sqsubseteq Refinement \\
MIDlet &\sqsubseteq Refinement \\
AssociationAttributes &\sqsubseteq Java1Refinement \\
Java2AssociationAttributes &\sqsubseteq Java2Refinement \\
AsyncMethods &\sqsubseteq Refinement \\
UMLtoJava &\sqsubseteq Refinement \\
DataTypes &\sqsubseteq Java1Refinement \\
Java2DataTypes &\sqsubseteq Java2Refinement \\
Observer &\sqsubseteq Refinement \\
JavaObserver &\sqsubseteq Refinement \\
Singleton &\sqsubseteq Refinement \\
RefinementConfiguration &\sqsubseteq Refinement \\
CompleteRefinementConfiguration &\sqsubseteq RefinementConfiguration \\
CompleteRefinementConfiguration &\equiv \exists\, implies\ UMLtoJava \\
&\sqcap \exists\, implies\ (Accessors \sqcup Java2Accessors) \\
&\sqcap \exists\, implies\ (Applet \sqcup MIDlet) \\
&\sqcap \exists\, implies\ (AssociationAttributes \sqcup \\
&\quad Java2AssociationAttributes) \\
&\sqcap \exists\, implies\ AsyncMethods \\
&\sqcap \exists\, implies\ (DataTypes \sqcup Java2DataTypes) \\
&\sqcap \exists\, implies\ (JavaObserver \sqcup Observer) \\
&\sqcap \exists\, implies\ Singleton \\
InvalidRefinementConfiguration &\sqsubseteq RefinementConfiguration \\
ImpliesInvalidConfiguration &\sqsubseteq InvalidRefinementConfiguration \\
ImpliesInvalidConfiguration &\equiv \exists\, implies\ InvalidRefinementConfiguration \\
Java1AndJava2 &\sqsubseteq InvalidRefinementConfiguration \\
Java1AndJava2 &\equiv \exists\, implies\ Java1Refinement \\
&\sqcap \exists\, implies\ Java2Refinement \\
AppletAndMIDlet &\sqsubseteq InvalidRefinementConfiguration \\
AppletAndMIDlet &\equiv \exists\, implies\ Applet \\
&\sqcap \exists\, implies\ MIDlet \\
ObserverAndJavaObserver &\sqsubseteq InvalidRefinementConfiguration \\
ObserverAndJavaObserver &\equiv \exists\, implies\ JavaObserver \\
&\sqcap \exists\, implies\ Observer
\end{aligned}
$$

[2] The abbreviated syntax of the Protégé ontology editor (`http://protege.stanford.edu`) is used.

Note that only the refinement transformations are covered in this ontology. A separate ontology has been defined for the features that are specific to the Instant Messaging product line, which includes references to this ontology. This separate ontology is not shown here due to size constraints.

The ontology uses the "implies" property to determine which refinement transformations are included in the transformation. This property is transitive, which allows refinement transformations that are implied by other refinement transformations to be transitively included in the global refinement configuration.

The "CompleteRefinementConfiguration" concept is equivalent to a logic expression, which allows for automatically determining its instances. This logic expression states which refinement transformations should at least be implied to have a complete configuration. The sub-concepts of "InvalidRefinementConfiguration" again use logic expressions, this time to express the conditions for an invalid combination of refinement transformations. For each instance of "RefinementConfiguration" one can check whether it is complete and/or invalid. If neither can be inferred, it is an incomplete configuration. If both are inferred, it is an inconsistent configuration. In OWL-DL, it is also possible to define annotations for each element. Such annotations can be used to define external (context) constraints for each refinement transformation.

Note that the order in which transformations have to be executed cannot be efficiently defined within the chosen concept structure. An alternative structure can be defined that uses separate properties to chain the refinement transformations together. For example, the following property can be used to chain AssociationAttributes and Accessors together:

$$impliesAccessors : AssociationAttributes \times Accessors$$

Such a property cannot be made transitive, because the domain and range are different. Because of this, completeness and incorrectness concepts will have to be defined separately for each refinement transformation concept.

Since OWL is built on top of RDFS [21], OWL supports annotation through RDFS. Therefore, OWL-DL supports attaching external constraints such as platform dependencies.

5.2 Domain-Specific Languages

Using the DSL technique, one can define a grammar or meta-model that describes the legal configurations of model transformations. Below is an EBNF grammar for our running example:

```
InstantMessagingClient  = 'InstantMessagingClient' UserInterface+
                          Network+ RefinementConfiguration .
UserInterface           = 'AWT' | 'Swing' | 'LCDUI' .
Network                 = JabberNetwork | 'Local' | 'SMS' .
JabberNetwork           = 'Jabber' JabberTransport .
JabberTransport         = 'DefaultJabber' | 'MEJabber' .
```

```
RefinementConfiguration = AssocAttrRefinement
                          AccessorsRefinement
                          ObserverRefinement
                          AppletRefinement
                          SingletonRefinement
                          AsyncMethodsRefinement
                          DataTypesRefinement
                          CodeGenerator .
AssocAttrRefinement     = ‘AssociationAttributes’ |
                          ‘Java2AssociationAttributes’ .
AccessorsRefinement     = ‘Accessors’ | ‘Java2Accessors’ .
ObserverRefinement      = ‘Observer’ | ‘JavaObserver’ .
AppletRefinement        = ‘Applet’ | ‘MIDlet’ .
SingletonRefinement     = ‘Singleton’ .
AsyncMethodsRefinement  = ‘AsyncMethods’ .
DataTypesRefinement     = ‘DataTypes’ | ‘Java2DataTypes’ .
CodeGenerator           = ‘UMLtoJava’ .
```

The part up to `RefinementConfiguration` is specific to the Instant Messaging product line. The part after describes the configuration rules for the refinement transformations in general. Note that EBNF does not provide a means to separate the specific part of the grammar from the general part. The grammar allows for configurations that have one or more user interfaces, one or more network protocols and exactly one refinement configuration. It is not possible to express that there may only be one user interface of each kind in an efficient way (without dropping the “at-least-one-user-interface” constraint). Similarly, the valid combinations of refinement transformations cannot be efficiently expressed (see sub-section 5.1). In such a case, transformations can be used to explicitly validate a configuration using additional configuration rules. The ASF+SDF grammar framework provides a rule-based rewriting system for this purpose.

An alternative for defining an EBNF grammar for our DSL is to define a meta-model. The Eclipse Modeling Framework (EMF) [8] provides the Ecore language that allows one to define and use meta-models. Fig. 2 shows a graphical representation of the meta-model that describes the configuration rules for the instant messaging client. This meta-model refers to another meta-model for the “RefinementConfiguration”. The meta-model that describes the configuration rules for the refinement transformations is shown in Fig. 3. Only part of this meta-model is shown due to space constraints.

EMF allows for defining annotations for each element. For our example, we have used annotations to associate platform dependencies to the meta-classes. These platform dependencies are defined in a separate model [18]. In our example, the “InstantMessagingClient” meta-class (see Fig. 2) has an annotation “InstantMessengerConstraints.owl#InstantMessagingClientPlatform”. This annotation points to the “InstantMessagingClientPlatform” element in the “InstantMessengerConstraints.owl” model. Similarly, the “AssociationAttributes” meta-class (Fig. 3) has an annotation to the “AssociationAttributesPlatform” element in separate platform constraints model for the refinement transformations. Note that there is also a “ContextConstraint” annotation in Fig. 3. All context constraint annotations point to this annotation to indicate they are of the context constraint “type”.

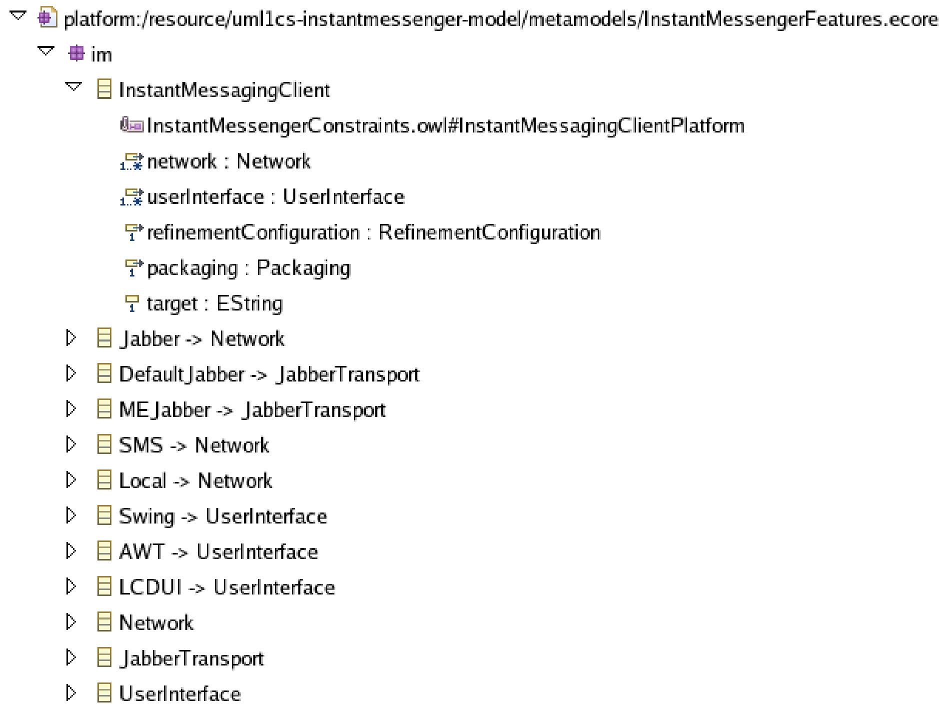

Fig. 2. EMF meta-model of the instant messaging client features

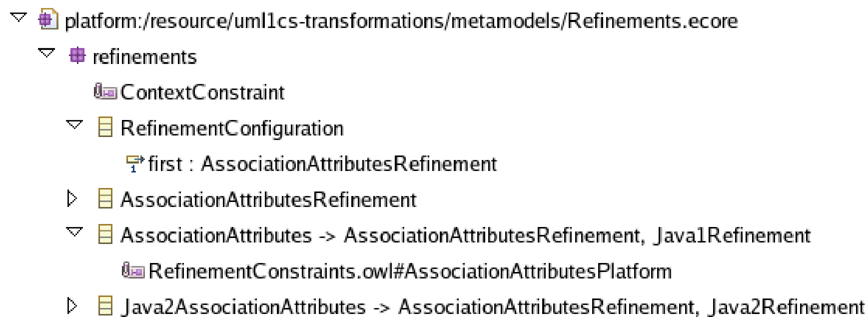

Fig. 3. Part of the EMF meta-model of the refinement model transformations

Similarly to grammars, Ecore meta-models also cannot express certain constraints in an efficient way. One can use OCL expressions to define extra constraints or one can define a model transformation that explicitly checks the extra constraints. In the case of Ecore, ATL can be used to transform the model into a “true” or “false” statement, indicating either a valid or an invalid model. In addition, it is harder to enforce sequence for the refinement transformations,

because meta-modelling is closer to a graph structure than a tree structure. The only way to enforce sequence in the meta-model is to use nesting of model elements, since the nesting follows a tree structure again. An alternative way of enforcing the sequence is to encode it in a model transformation. Such a model transformation could be used to place the chosen refinement transformations in the correct order.

5.3 Feature Modelling

Feature models are generally visualised with feature diagrams. Fig. 4 shows a partial feature diagram for our running example. Not all features are shown due to space constraints.

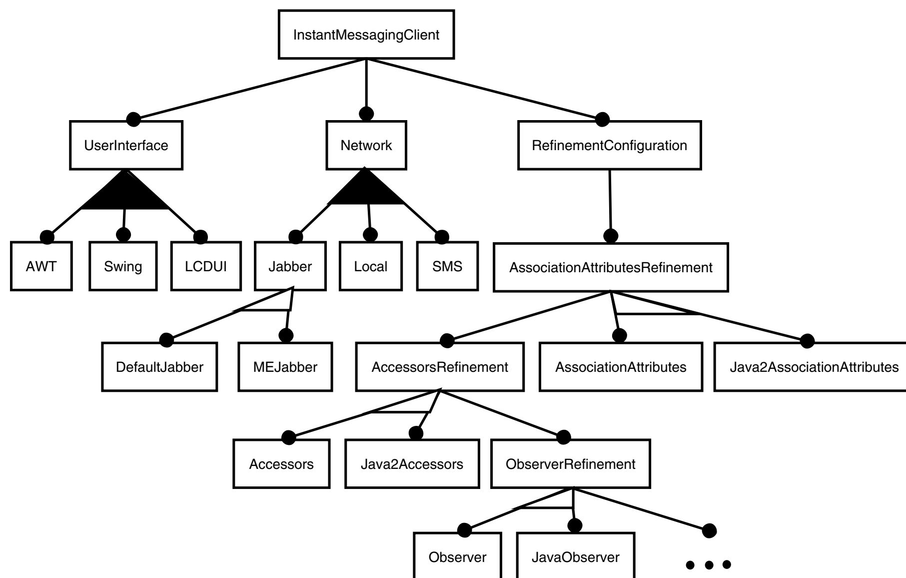

Fig. 4. Partial feature diagram of the instant messaging client features

Similar to meta-modelling, a tree structure is necessary to express sequence. Note that it is possible to express the constraints of having at least one user interface, yet any of them may occur only once. This is due to the nature of feature diagrams in which features can occur only once by default. Extensions to feature diagrams exist that allow for expressing feature multiplicities [14]. It is not possible to efficiently express the additional constraints on valid refinement transformation combinations. The Feature Modeling Plugin tool[3] solves this problem by allowing XPath constraints to be defined. Note that this solution works on the XML representation level, not on the feature diagram level.

[3] http://gp.uwaterloo.ca/fmp/

The Feature Modeling Plugin also allows for annotations, such that external constraints can be attached.

6 Comparison and Discussion

This section compares the different configuration techniques for each criterion listed in section 4. Table 1 gives an overview of how well each technique fulfils the requirements. The scale is divided into five steps, ranging from -- to ++.

Regarding modularity, meta-modelling provides the best mechanism, since it explicitly supports modules of language elements (i.e. packages and meta-models). The OWL-DL knowledge-based approach supports different ontology namespaces, but this is implemented on the level of RDFS. Grammars and feature models have no explicit support for modularity. This can be added by building on top of a language – and corresponding tool – that supports modularity. The Feature Modeling Plugin, for example, is built on top of EMF's Ecore language. This way, the Feature Modeling Plugin can use EMF to access elements of the feature model. This mechanism works the same for local and inter-model references.

OWL-DL is designed for reusability: OWL-DL allows for the definition of common vocabularies with particular semantics. Each ontology that builds on top of that vocabulary inherits those semantics. Meta-modelling does not provide real reuse by itself. EMF supports reuse of meta-models through its XMI [22] layer. Feature modelling more or less provides reusability through staged configuration: specialisations of feature models are derived from a general feature model to outline the scope for a product line subset. Grammars offer no native support for reuse, but this can again be added by building the grammar on top of a language that does support reuse.

If one includes the transformation engines for DSLs (both for grammars and meta-modelling), the expressiveness is beyond what is necessary for our purposes. The expressiveness of the OWL-DL knowledge-based system is sufficient, but is inefficient for expressing sequence. The expressiveness of feature models is also sufficient, but inefficient for expressing more complex constraints.

Table 1. Comparison of configuration techniques

Techniques \ Criteria	modularity	reusability	expressiveness	maintainability	locality	extensibility
Knowledge-based systems	+	++	+	-	+/-	++
DSL - grammar	-	-	++	+	+/-	+/-
DSL - meta-modelling	++	+	++	+	+/-	++
Feature modelling	-	+/-	+	-	+	+

The inefficiency in expressing certain constraints can cause reduced maintainability for both the knowledge-based approach as well as feature modelling. A change in these constraints may require extensive refactoring of the configuration rules. This is not such an issue for the DSL-based techniques, which allow complex constraints to be expressed in a separate constraint checking transformation.

The locality principle requires that configuration constraints do not require global checking for a local modification of the configuration. This principle inherently holds for feature modelling, since constraints can only apply to the directly neighbouring elements. For all other techniques, the user of that technique is responsible for keeping the complexity of the constraints in check. If one constraint requires the checking of too many related constraints, the constraint checking will not scale sufficiently.

Extensibility, in this context, refers to the possibility to annotate the configuration rules with external constraints (e.g. platform dependencies), such that the configuration system can be "extended" with an extra constraint checking mechanism. Both OWL-DL and meta-modelling are built on top of languages that provide support for annotations. Feature models in general do not provide annotation support, but the Feature Modeling Plugin implementation does. The Feature Modeling Plugin annotations can be accessed through the underlying modelling framework. EBNF grammars in general also do not provide annotation support, but most grammar frameworks support comments. These comments are not as easily accessible through the grammar framework, however (one must usually know the navigation path to these comments).

7 Conclusion

In this paper, we have introduced and compared a number of configuration techniques for the purpose of configuring model transformations in an MDA context. Based on a number of specific model transformation properties, several comparison criteria have been derived. The comparison shows the feasibility of each configuration technique for each of these criteria. The paper then discusses which aspects of the configuration techniques contribute to the required criteria.

In the discussion of DSLs, the language definition formalism is used to describe a particular set of model transformations. DSL techniques can also be used to define a "Transformation Configuration DSL". This allows for representing concepts like "platform dependency" as first-class language elements instead of annotations. If "model transformation" and "platform dependency" are part of the language definition, the particular instances of model transformations and their platform dependencies become an expression in that language. Configurations of transformations again contain occurrences (i.e. instances) of the various model transformation instances. This is commonly addressed by adding an explicit "instanceOf" relationship to the language definition (examples are UML and OWL). Of course, when adding your own "instanceOf" relationship, you also have to provide the semantics for it. This usually boils down to providing

your own tool support to check whether your particular configuration of model transformation occurrences complies with the configuration rules you defined for the model transformations. This comes close to a "DSL-within-a-DSL" and leads to the discussion of whether we need a special language definition formalism for "Transformation Configuration". Such a special language definition formalism is considered out of the scope of this comparison, since it does not yet exist.

In the comparison, we did not explicitly consider step-wise refinement [23]. This is a paradigm for developing a complex program from a simple program by adding features incrementally. In the AHEAD tool an arbitrary number of programs and features is expressed as nested sets of equations. This model uses algebraic specifications, which can be reduced to a DSL approach that provides a language definition formalism and an algebra that operates on this formalism.

Acknowledgement

The authors would like to thank the anonymous review committee for their comments, which included some very useful points for consideration and discussion.

References

1. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electr. Notes Theor. Comput. Sci. **127** (2005) 113–128
2. Stefik, M.: Introduction to Knowledge Systems. Morgan Kaufmann Publishers Inc. (1995)
3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. The SEI Series in Software Engineering. Addison Wesley Professional (2001)
4. McDermott, J.: XSEL: a computer sales person's assistant. In J.E.Hayes, Michie, D., Y-H.Pao, eds.: Proceedings of the Tenth Machine Intelligence Workshop, held at Case Western Reserve University, Cleveland, USA, Ellis Horwood (1982) 325–338
5. Deursen, A.v., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices **35** (2000) 26–36
6. Brand, M.v.d., Deursen, A.v., Heering, J., Jong, H.A.d., Jonge, M.d., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In Wilhelm, R., ed.: Proceedings of the 10th International Conference on Compiler Construction (CC 2001). Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. Volume 2027 of Lecture Notes in Computer Science., Springer-Verlag (2001) 365–370
7. Object Management Group, Inc.: Meta Object Facility (MOF) 2.0 Core Specification. (2003) Version 2.0, Available Specification, ptc/04-10-15.
8. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. The Eclipse Series. Addison Wesley Professional (2003)
9. Bravenboer, M., Dam, A.v., Olmos, K., Visser, E.: Program Transformation with Scoped Dynamic Rewrite Rules. Fundamenta Informaticae **69** (2005) 1–56

10. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Model Transformations in Practice Workshop at MoDELS 2005, Montego Bay, Jamaice. (2005)
11. Object Management Group, Inc.: OCL 2.0 Specification. (2005) Version 2.0, ptc/2005-06-06.
12. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA (1990)
13. Coplien, J., Hoffman, D., Weiss, D.: Commonality and Variability in Software Engineering. IEEE Software **15** (1998) 37–45
14. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10** (2005) 7–29 Special Issue on Software Variability: Process and Management.
15. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice **10** (2005) 143–169 Special Issue on Software Product Lines.
16. Deursen, A.v., Klint, P.: Domain-Specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology **10** (2002) 1–17
17. Reiff-Marganiec, S., Ryan, M., eds.: Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI2005), Leicester, UK, IOS Press (2005)
18. Wagelaar, D., Jonckers, V.: Explicit Platform Models for MDA. In: Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), Montego Bay, Jamaica. Volume 3713 of Lecture Notes in Computer Science., Springer-Verlag (2005) 367–381
19. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P., eds.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press (2003)
20. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. World Wide Web Consortium. (2004) W3C Recommendation 10 February 2004, [Online] `http://www.w3.org/TR/owl-guide/`.
21. Dan Brickley, R.G.: RDF Vocabulary Description Language 1.0: RDF Schema. World Wide Web Consortium. (2004) W3C Recommendation 10 February 2004, [Online] `http://www.w3.org/TR/rdf-schema/`.
22. Object Management Group, Inc.: MOF 2.0/XMI Mapping Specification. (2005) Version 2.1, formal/05-09-01.
23. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering **30** (2004) 355–371