

Context-Driven Model Refinement

Dennis Wagelaar*

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

Abstract. An important drive for Model-Driven Architecture is that many software applications have to be deployed on a variety of platforms and within a variety of contexts in general. Using software models, e.g. described in the Unified Modeling Language (UML), one can abstract from specific platforms. A software model can then be transformed to a refined model, given the context in which it should run. Currently, each target context requires its own model transformation. Only a limited number of contexts can be supported in this way. We propose a context-driven modelling framework that models each target context in a context model, described in the Web Ontology Language (OWL). Multiple reusable transformation rules are used, which are annotated with context constraints, based on the OWL context model. The framework can automatically select the transformation rules that are applicable for a concrete context.

1 Introduction

The Model-Driven Architecture (MDA) allows for mapping a high-level software design to a specific implementation platform. Model transformations are used to refine a Platform-Independent Model (PIM) to a Platform-Specific Model (PSM). Several layered PSMs can be used to gradually refine the design.

An important drive for MDA is that a lot of software has to run within a variety of computing contexts. The vision of Ambient Intelligence only increases this variety, with many portable and embedded devices such as personal digital assistants (PDAs), smartphones and embedded computers in cars and houses. For our purposes, context includes the software/hardware platform on which the software must run, but also other factors, such as required run-time qualities (e.g. adaptability, performance, security, etc.) and user preferences (e.g. chosen software features).

In current MDA approaches [1], each target platform requires its own (set of) model transformation(s). This means that for each new target platform, at least one new model transformation is needed, even if that platform is only a variant of another, already supported platform (e.g. J2ME Personal Profile¹ is a variant

* 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)

¹ <http://java.sun.com/j2me/>

of Java). In practice, this means that only a relatively small number of general platforms can be targeted, e.g. Java or C++. Targeting very specific platforms, e.g. the previously mentioned J2ME Personal Profile 1.0, is not feasible because of the maintenance overhead, even though such precise targeting can result in PSM that is better optimised for the given context.

Looking into the model transformations themselves, it appears that they can often be made reusable over multiple platforms and it is only how they are configured that makes them applicable to only one platform. For example, one model transformation could target all Java 2 platforms by transforming “to-many” association ends in the Unified Modeling Language (UML) to attributes using the Java 2 Collections framework. If this transformation is configured to be applied in combination with a transformation that targets the Java Swing framework, the target platform is already limited to J2SE for the desktop computer². The fact that each configuration of model transformations is also maintained by hand, makes that the problem of limited platform support remains.

We propose a context-driven modelling framework that can automatically select appropriate transformation rules for a concrete context and configure them into a context-optimised transformation. The developer can define a number of alternative refinement transformation rules. These transformation rules are annotated with context constraints within which the transformation will work. These context constraints, as well as the concrete context description, are based upon an explicit context model. This context model is expressed in the Web Ontology Language (OWL) [2], which is an extensible language for describing ontologies. Furthermore, we use the OWL-DL variant, which corresponds to description logics [3], such that computational completeness can be guaranteed. The context model forms a basis for describing contexts in general and can be extended to include the specific context information that is relevant for a particular application domain. An automatic reasoner, such as RACER [4], can be used to verify whether a concrete context satisfies the constraints of a model transformation. Subsequently, one transformation rule is selected for each group of remaining transformation alternatives, based on how close its constraint lies to the actual context.

Section 2 discusses how computing context can be modelled and section 3 explains how model transformations can be augmented with context constraints. In section 4 the mechanism for selecting model transformations is explained. Section 5 discusses related work and section 6 states the conclusions for this paper.

2 Context Modelling

In order to reason about context and context constraints, an ontology of computing context is used. Ontologies can serve as a common vocabulary for a domain [5]. The relationships between the ontology elements can be used to reason about elements based on that ontology. A context ontology allows one to

² <http://java.sun.com/j2se/>

base expressions about a concrete context on the vocabulary expressed by the ontology. By using a shared model of context, we can reason about the relationship between a context description and a context constraint, even if the two do not have a direct relationship. An example context constraint is that the Java 2 Collections framework needs to be present. An example of a context description includes a Sharp Zaurus PDA. Since both the context constraint and the context description refer to the context ontology to explain what the Java 2 Collections framework resp. the Zaurus PDA is, one can derive whether the Zaurus PDA satisfies the Java 2 Collections framework constraint.

2.1 A Context Vocabulary

Before modelling any specific context or context properties, a basic context structure needs to be defined. In this paper, the context ontology described in [6] is used for this purpose³. This ontology is in turn inspired by the User Agent Profile specification (UAProf) [7] and Composite Capability/Preference Profiles (CC/PP) [8], both of which are standards intended to describe target platforms. The ontology is expressed in OWL, an extensible standard for describing ontologies. OWL has a variant, called OWL-DL, that corresponds to description logics, allowing for automated reasoning about the ontology. The ontology used complies to this OWL-DL variant. The part of the ontology that models platforms and users is shown in Fig. 1.

The platform concept in this ontology can provide software and hardware. The '*' next to the "providesSoftware" relationship denotes a one-to-many relationship. Software and hardware are broken down into different sub-concepts. This is denoted by the special "isa" subsumption relationship, e.g. the set of operating systems subsumes the set of software in general. The software can impose requirements on the platform, e.g. the need for a network resource, a particular virtual machine or a user interface rendering engine that supports voice communication. This is denoted by the "requiresPlatform" relationship, which points to a description of the required platform. The user concept has profile elements, amongst others, which describes the user. A special case of a profile is a preference profile, which describes user preferences only.

The ontology can be extended for particular domains of platforms, such as Java virtual machines. Fig. 2 shows such an ontology. The "VirtualMachine" concept starts with "context:" to indicate it refers to the "VirtualMachine" concept from the main context ontology. The "Java" virtual machine can be subdivided in many different configurations. "Java2" refers to the virtual machines that run Java 1.2 or up. "Java2" is split up into "J2ME", "J2SE" and "J2EE", which is based on "J2SE". In the ontology, two other concepts are introduced: "AWT" and "Swing". These refer to the Java Abstract Window Toolkit (AWT) resp. Swing rendering engines. Since some Java virtual machine configurations already include these, instances of such virtual machines also serve as instances

³ Other ontologies can be used, but it is necessary to use the same ontology for describing context and constraints on that context.

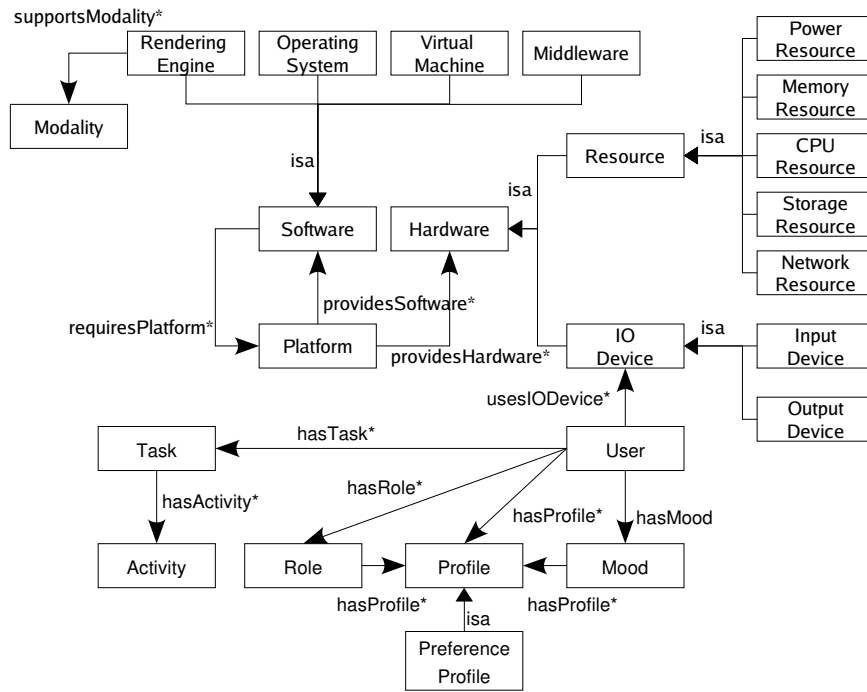


Fig. 1. Part of the context ontology for describing platforms and users

of the AWT and/or Swing rendering engine. This is represented in the ontology by defining additional “isa” relationships to the AWT or Swing rendering engine from the virtual machines.

In order to discriminate on user profile data as well, an ontology extension for user profiles can be defined. For the purpose of this paper, we will use a very simple profile ontology, describing only the languages that a user can use and/or prefers to use. This ontology extension is depicted in Fig. 3.

2.2 Modelling Concrete Contexts

The actual context for the PSM can now be described by instances of the context concepts. Fig. 4 shows an example context description for a Sharp Zaurus PDA. This PDA has a Personal Profile (PP) J2ME virtual machine installed.

The concepts “Platform” and “PP” are taken from the context resp. Java ontologies. The instances, “zaurusC860” and “zaurusPP”, are depicted as rounded rectangles and are instances of the “Platform” and “PP” classes. This is depicted by the “io” relationships. The “zaurusC860” platform has a “providesSoftware” relationship with the “zaurusPP” Java Personal Profile virtual machine. The “zaurusC860” platform provides a “touchscreen” and a “keyboard” I/O device,

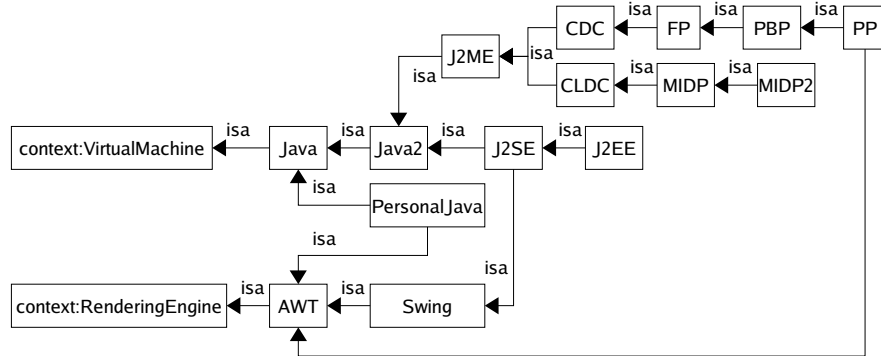


Fig. 2. An ontology describing Java virtual machines

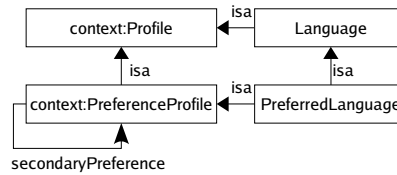


Fig. 3. An example ontology for describing user profiles

through which the user “dennis” interacts with the platform. Finally, “dennis” prefers to communicate in “dutch”.

3 Modelling Context Dependencies

Model transformations can now define constraints on instances of the ontology concepts. This is done by defining new, *completely specified* concepts. Such concepts have *necessary and sufficient* conditions in addition to any *necessary* conditions. For example, whereas it is *necessary* that each “J2ME” instance is also an instance of “Java2” (depicted by the “isa” relationship in Fig. 2), being a “Java2” instance is not *sufficient* for also being a “J2ME” instance. The notation for describing conditions as used in the Protégé tool [9] is also used here.

In order to check if the current platform has a “Java2” class virtual machine, a concept “Java2Platform” can be defined, which is a sub-concept of “Platform” (*necessary*) and provides a “Java2” virtual machine (*necessary and sufficient*):

$$\begin{aligned}
 \text{Java2Platform} &\sqsubseteq \text{context : Platform} \\
 &\equiv \exists \text{context : providesSoftware platform : Java2}
 \end{aligned}$$

Whenever a “Platform” instance fulfils the condition of providing a “Java2” virtual machine, it can be classified as an instance of “Java2Platform”. This

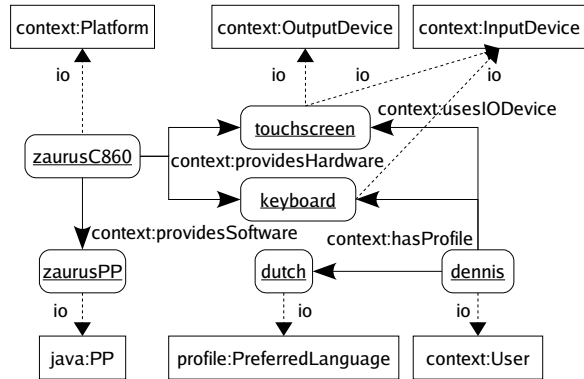


Fig. 4. Example context description for the author using the Sharp Zaurus PDA

classification can be performed by automatic reasoners. This way, concrete platform instances can be matched against a completely defined constraint concept. If the platform instance classifies as an instance of the constraint concept, then the constraint holds for that instance. For example, the “zaurusC860” platform from Fig. 4 classifies as an instance of “Java2Platform”, since “zaurusPP” is an instance of “PP”, which is a sub-class of “Java2”.

3.1 Example PIM

Fig. 5 shows part of a PIM for a simple Breakout game, expressed in UML 1.5 [10]. The objective of the game is to remove all the bricks from the screen by hitting them with the ball. The ball must be bounced back with the paddle, which is controlled by the player. If the ball falls down the screen (paddle has missed), the game is over. In the design, the “Field” class represents the screen, which has composition associations with a “Ball”, a “Paddle” and multiple “Bricks”. A separate “view” package has been modelled to separate the graphical user interface from the game model itself. AWT and Swing implementations of the “view” package have also been modelled, but are not shown in the diagram. Note that our example PIM contains platform-specific elements that rely on the AWT and Swing rendering engine. Our notion of PIM includes all models that have not (yet) committed to a specific platform.

The example PIM contains several elements that are not available in the programming environment that is needed for the target platform⁴. These elements are the “process”, “Observer”, “Observable”, “subscribe” and “thread” stereotypes, the “Integer” and “Boolean” Object Constraint Language (OCL) data types, association relationships and specifications of operations (e.g. in OCL, a dynamic diagram or an Action Language). Model transformations can

⁴ The programming environment comprises the programming language and available libraries

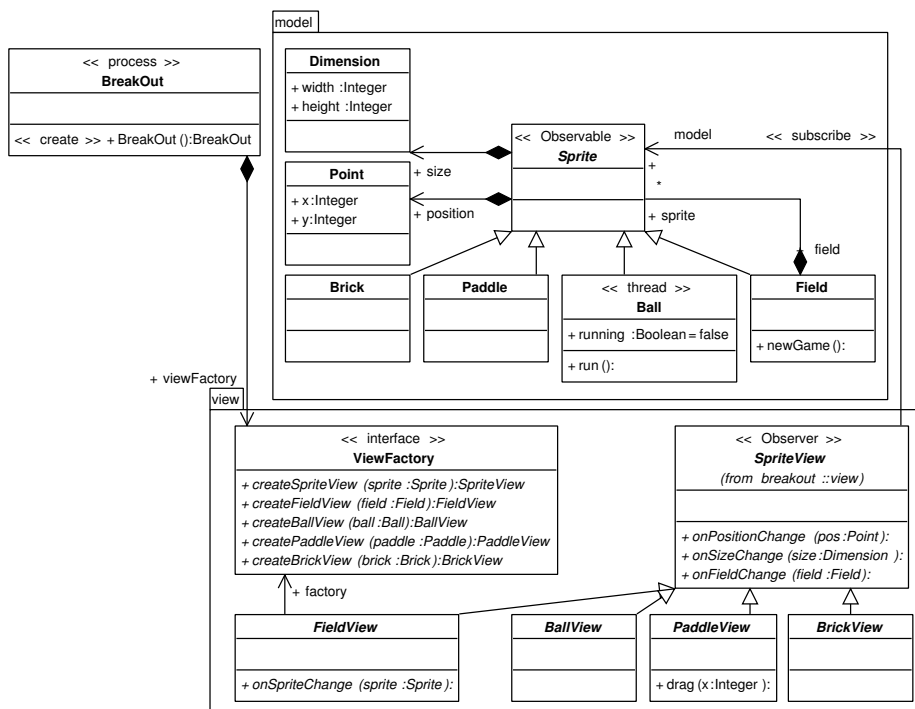


Fig. 5. Example PIM class diagram for a breakout game

be defined to translate each of these elements to one or more elements that are available in the target programming environment. In addition, the PIM contains several platform-specific elements, such as the view implementations relying on Java AWT and Swing. The selection of relevant platform-specific elements can also be performed by model transformations. Examples of some of these model transformations will be discussed below. The Atlas Transformation Language (ATL) [11], which has a simple, rule-based syntax, will be used to express these examples.

3.2 Example Model Transformations

A model transformation for translating UML 1.5 associations to corresponding attributes for Java could use the Java 2 Collections framework to implement a one-to-many association. The following transformation rules use the `java.util.List` interface and the implementing `java.util.ArrayList` class to achieve a one-to-many relationship⁵:

⁵ Note that, in ATL, additional headers are needed and rules are necessary for each model element that needs to be copied/transformed. Only the rules that perform actual transformation are shown here for brevity.

```

rule AssociationEndSingleAttribute {
  from s : INMODEL!AssociationEnd (s.isNavigable and s.isSingle())
  to t : OUTMODEL!Attribute (
    name <- s.name,
    owner <- s.navigableFrom(),
    type <- s.participant,
    visibility <- s.visibility,
    ownerScope <- s.targetScope,
    changeability <- s.changeability)
}

rule AssociationEndArrayList {
  from s : INMODEL!AssociationEnd (s.isNavigable and not s.isSingle())
  using { collection : INMODEL!Interface = INMODEL!Interface.allInstances()
    ->select(c|c.name='Collection')->first(); }
  to t : OUTMODEL!Attribute (
    name <- s.name,
    owner <- s.navigableFrom(),
    type <- collection,
    visibility <- s.visibility,
    ownerScope <- s.targetScope,
    changeability <- s.changeability,
    initialValue <- v),
  v : OUTMODEL!Expression (
    language <- 'java',
    body <- 'new java.util.ArrayList()')
}

helper context INMODEL!AssociationEnd def : navigableFrom() :
  INMODEL!Classifier =
  self.association.connection->select(x|x<>self)->first().participant;

helper context INMODEL!AssociationEnd def : isSingle() : Boolean =
  self.multiplicity.range->select(r|r.upper<>1)->isEmpty();

```

The transformation rules translate only the navigable association ends to attributes. The first rule translates all association ends with an upper multiplicity range of “1” to simple attributes. The second rule translates all association ends with an upper range other than “1” to Java Lists. The `from` keyword indicates the element to read from the source model, whereas the `to` keyword indicates the element to be created in the target model. The `INMODEL` and `OUTMODEL` in the transformation refer to the meta-models used, which is the UML 1.5 meta-model in both cases. Two `helper` functions have been defined to reuse the OCL expressions for determining the class from which the association end can be navigated (`navigableFrom()`)⁶ and whether an association end only points to a single target (`isSingle()`). The second rule has a `using` clause, which locates the Java Collection interface. This Collection interface is then used as the type of the attribute that is created. The `ArrayList` class is used for the initial value of this attribute.

The `AssociationEndSingleAttribute` transformation rule does not use any Java-related elements, and has no platform dependencies. The `AssociationEndArrayList` rule uses the Java 2 Collections framework and therefore needs at least a “Java2” virtual machine (see Fig. 2). This corresponds with the “Java2-Platform” constraint given at the beginning of this section. An alternative for the `AssociationEndArrayList` rule could use the `java.util.Vector` class to implement the one-to-many association:

⁶ only binary associations are considered

```

rule AssociationEndVector {
  from s : INMODEL!AssociationEnd (s.isNavigable and not s.isSingle())
  using { vector : INMODEL!Class = INMODEL!Class.allInstances()
        ->select(c|c.name='Vector')->first(); }
  to t : OUTMODEL!Attribute (
    name <- s.name,
    owner <- s.navigableFrom(),
    type <- vector,
    visibility <- s.visibility,
    ownerScope <- s.targetScope,
    changeability <- s.changeability,
    initialValue <- v),
  v : OUTMODEL!Expression (
    language <- 'java',
    body <- 'new java.util.Vector()' )
}

```

Because the Java Vector class was already available in Java 1.0, the platform constraint can be relaxed to only requiring a Java virtual machine:

$$\begin{aligned}
JavaPlatform &\sqsubseteq context : Platform \\
&\equiv \exists context : providesSoftware\ java : Java
\end{aligned}$$

4 Context-Driven Refinement

The mechanism that selects the appropriate model transformations is based on Synthesis-Based Design [12] and its version for MDA transformations [13]. Synthesis-Based Design uses a *design space* of possible combinations of alternative design choices. The design choices are represented by model transformations in this case. The transformations are grouped into sets of alternatives that represent the same functionality. This grouping can be done automatically, based on a heuristic that checks the input specification of the transformation rules. If certain transformation rules have the same input specification, they are considered to be alternatives. The transformation rules `AssociationEndArrayList` and `AssociationEndVector`, given in subsection 3.2, have the same input specification (represented by the `from` part). Hence, they are considered to be alternatives belonging to one group. The groups that are formed in this way only have to be created once for a set of transformation rules and can be adapted manually afterwards. Each time a PSM has to be generated for a specific platform, the existing transformation rules are considered, using the existing grouping information. An example grouping for the model transformations needed for our example PIM is shown in Table 1.

The `English`, `French`, `German` and `Dutch` transformations are simple selection transformations that select their corresponding language package to be included in the deployment. They form one group, since they take all language package deployment information as input and differ only in which language packages are copied back out. Similarly, the `AWTView` and `SwingView` transformations select the AWT resp. Swing view implementation from all view implementations. The `AssociationEndArrayList` and `AssociationEndVector` transformations have already been discussed and form one group. The `Accessors`

English French German Dutch
AWTView SwingView
Accessors
AssociationEndArrayList AssociationEndVector
Process
Thread
Observer PropertyChangeListener
DataTypes

Table 1. Example model transformations grouping

transformation creates accessor operations (getters and setters) for each public attribute. The **Process** transformation augments all classes with the “process” stereotype with a “main” operation. The **Thread** transformation adds a realization relationship to the `java.lang.Runnable` class to each class with the “Thread” stereotype. The **Observer** and **PropertyChangeListener** transformations both implement the “Observer”, “Observable” and “subscribe” stereotypes. The first transformation uses the Java 1.0 `java.util.Observer` interface and the `java.util.Observable` class to accomplish this, while the latter uses the `java.beans.PropertyChangeListener` interface and corresponding classes. Finally, the **DataTypes** transformation translates the OCL data types into Java data types.

From each group, one model transformation is selected. First, all transformation constraints are checked against the platform, after which the non-matching transformations are discarded. Note that, if no transformations are left for a particular group after this step, no PSM can be generated for the given context. Each group of remaining alternative transformations is sorted by context relevance, such that the most relevant transformation alternative appears at the top of the list. The context relevance is determined by subsumption of constraint concepts. If a constraint concept defines a subset of another constraint concept, then that constraint is considered more context-specific. Consider the following context constraint:

$$\begin{aligned}
 \text{Java2Personal} &\sqsubseteq \text{context} : \text{Platform} \\
 &\equiv (\exists \text{context} : \text{providesSoftware } \text{java} : \text{Java2}) \sqcup \\
 &\quad (\exists \text{context} : \text{providesSoftware } \text{java} : \text{PersonalJava})
 \end{aligned}$$

This constraint demands either a “Java2” or a “PersonalJava” VM, whereas the “JavaPlatform” constraint (see before) demands a “Java” class VM. Both “Java2” and “PersonalJava” are subconcepts of “Java”. The set defined by the union of “Java2” and “PersonalJava” is still a subset of “Java”, so the “Java2-Personal” concept can be classified as a subconcept of “JavaPlatform”. Again, existing automatic reasoners can be used to classify the subsumption taxonomy of concepts as they are defined by the constraints.

It is possible that the context constraints of alternative transformations do not contain enough information to determine whether one constraint subsumes another. Consider the following example constraint:

$$\begin{aligned} \text{AWTPlatform} &\sqsubseteq \text{context} : \text{Platform} \\ &\equiv \exists \text{context} : \text{providesSoftware } \text{java} : \text{AWT} \end{aligned}$$

Compared to the “Java2Platform” constraint mentioned earlier, one cannot classify either as a subset of the other. In such a case, the group of transformation alternatives (see Table 1) will first be reduced to those alternatives of which the constraints are leaves in the constraint concept taxonomy. From these alternatives, the alternative specified left-most in the initial group will be chosen. Consider three transformation alternatives, A, B and C, which have the “JavaPlatform”, “Java2Platform” and “AWTPlatform” constraint respectively. If a taxonomy is created for these constraints, “Java2Platform” and “AWTPlatform” are both direct subconcepts of “JavaPlatform”. The group of alternatives is reduced to B and C, since their constraints are the leaves in the taxonomy. If alternative B is listed before alternative C in the initial group of alternatives, then alternative B will be chosen.

Since some model transformations may depend on the result of other model transformations, they need to be ordered. The transformation dependencies can also be checked automatically by a heuristic that checks if the input specification of a transformation may overlap with the output specification of another (represented by the `to` part). The output specification of the `AssociationEndArrayList` transformation states that it creates new attributes. If another model transformation, `Accessors`, creates accessor operations for each public attribute, then its input specification could match elements generated by `AssociationEndArrayList`. Hence, `AssociationEndArrayList` is placed before `Accessors`. If no decision can be made on whether to put one transformation before another, the order is left unchanged. This way, the developer can already pre-sort the groups of transformations manually and no manual intervention is needed for each context. The sorted list of chosen transformations for the example platform from Fig. 4 is shown in Table 2.

The `AssociationEndArrayList` transformation was chosen over the `AssociationEndVector` transformation, because it requires a “Java2” VM instead of any “Java” VM. For the same reason, the `PropertyChangeListener` transformation is chosen over the `Observer` transformation. Also, the transformations have been sorted according to input-output dependencies: the `Accessors` transformation has been placed after `AssociationEndArrayList`. The `Dutch` selection transformation takes in all kinds of elements and can also output all kinds of elements, so the sorting heuristic could not determine what to do with it. In this case, the developer knows that this rule does not depend on any transformation output, so it remains pre-sorted as the first transformation to execute. The other transformations don’t generate any elements that may be matched by the input specification of another transformation, so their order is also not adapted.

Dutch
AWTView
AssociationEndArrayList
Accessors
Process
Thread
PropertyChangeListener
DataTypes

Table 2. Example model transformations selected and sorted

5 Related Work

In Generative Programming [14] and Step-Wise Refinement [15], *features* and *feature models* are used to model a family of software systems instead of a single system. Features can be optional or mandatory for a software system, depending on the presence of other features. In our framework, features are implicitly generated or selected by model transformations, which are chosen based on context constraints. Feature models can be used to verify if the chosen transformations represent a valid set of features.

The lack of explicit platform models is discussed in [16]. The notion of *abstract platform* is introduced, which describes a set of elements to model a PIM against. This set of elements includes design artifacts that are available in a target platform (classes, interfaces) and design constructs that can be mapped to that platform (stereotypes, profiles), e.g. with model transformations. The goal of abstract platforms is to ease platform-independent modelling, whereas our context models are meant to decouple context information, which includes the platform, from model transformations.

In [17], platform selection rules are discussed, which allow for pre-selecting a number of target platforms. In that way, less platforms need to be supported. In our case, platform selection rules can be used to narrow down the amount of platform domain aspects (e.g. Java virtual machines) that need to be modelled for a particular application domain (e.g. instant messaging). This does not conflict with the envisioned ambient intelligence scenario that targets an open-ended infrastructure of unanticipated devices, since this is supported by in-depth modelling of platform domain aspects, not the amount of aspects that are modelled.

In [18], an infrastructure for combining UML models and ontologies is introduced. Such an infrastructure can be useful for a better integration of platform constraints into model transformation languages.

The KobrA method [19] is an approach for component-based product line engineering with UML. It uses pattern-based refinements for design elements. OO-Method [20] also introduces a pattern-based approach for design refinement

and code generation. PRISMA [21] is a modelling approach that can be used to model context data. Our approach differs in that it uses refinement alternatives, such that context-based optimisation is possible.

The Catalysis approach [22] is a UML-based development method for component-based systems. An important part of this method consists of refinement of the model elements. As such, our context-driven modelling framework can be used as a means to refine the model elements in a context-optimised way.

The Context Ontology Language (CoOL) [23] is an ontology-based context modelling approach, which uses the Aspect-Scale-Context (ASC) model where each aspect (e.g. spatial distance) can have several scales (e.g. kilometre scale or mile scale) to express some context information (e.g. 20). Chen et al. [24] propose a context broker architecture (CoBrA) using an ontology to describe persons, places and intentions. Gu et al. [25] present a service-oriented context-aware middleware (SOCAM) based on a context model with person, location, activity and computational entity (such as a device, network, application, service, etc.) as basic context concepts. Henricksen and Indulska [26] propose a context model that describes context based on several types of facts (e.g. sensed, static and profiled) subject to constraints and quality annotations. The context ontology used in this paper puts more focus on the platform description, which is central to MDA.

6 Conclusion and Future Work

This paper has introduced a context-driven modelling framework that can automatically choose the most context-specific model transformations from a set of alternatives. Instead of providing a set of alternative model transformations, multiple sets of alternative model transformations are provided, which together can form a complete model transformation. In this way, many more computing contexts can be supported with a similar design effort.

The proposed modelling framework fits within the MDA vision in that it also uses several layered refinement transformations. Based on a context model described in OWL, specific transformations are chosen to transform a PIM to a PSM.

The selection mechanism relies on the classification of a taxonomy of context constraints. This classification needs to be done only once for a set of available model transformations and can then be reused for each concrete context. Furthermore, the constraint checking mechanism implemented by RACER is highly optimised. It should scale no worse than the matching algorithm needed for the model transformations themselves.

In the future, a configuration language will be introduced to support the transformation selection and sorting mechanism. This configuration language will express the inter-dependencies of the model transformations and will discriminate between mandatory (e.g. an accessor method generator) and non-mandatory transformations (e.g. a language support selection transformation). This configuration language will probably be based upon feature models. MOF

can be used for the description of the abstract syntax, such that the same repository that is used for storing the various MDA models can also be used for storing the configuration model.

Acknowledgement

The author would like to thank Ragnhild van der Straeten and the anonymous review committee for reviewing this paper. Many improvements have been made based on their comments. In addition, the author would like to thank Willem Hajenius and Wouter Heyse for their work on the breakout game example case used in this paper.

References

1. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
2. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. World Wide Web Consortium. (2004) W3C Recommendation 10 February 2004.
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, Cambridge, UK (2003)
4. Möller, R., Haarslev, V.: Description Logics for the Semantic Web: Racer as a Basis for Building Agent Systems. *Künstliche Intelligenz* (2003) 10–15
5. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition* **5** (1993) 199–220
6. Preuveneers, D., den Bergh, J.V., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V., Bosschere, K.D.: Towards an extensible context ontology for Ambient Intelligence. In: Proceedings of the Second European Symposium on Ambient Intelligence, Eindhoven, The Netherlands, Springer-Verlag (2004) 148–159
7. Open Mobile Alliance: User Agent Profile 2.0 Specification. (2003) Version 20-May-2003.
8. Klyne, G., Reynolds, F., Woodrow, C., Ohto, H., Hjelm, J., Butler, M.H., Tran, L.: Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. World Wide Web Consortium. (2004)
9. Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W., Musen, M.A.: Creating Semantic Web Contents with Protege-2000. *IEEE Intelligent Systems* **16** (2001) 60–71
10. Object Management Group, Inc.: Unified Modeling Language Specification. (2003) Version 1.5 (formal/03-03-01).
11. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In: OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
12. Tekinerdoğan, B., Akşit, M.: Synthesis Based Software Architecture Design. In Akşit, M., ed.: *Software Architectures and Component Technology*, Dordrecht, The Netherlands, Kluwer Academic Publishers (2001) 143–173

13. Kurtev, I., van den Berg, K.: A Synthesis-Based Approach to Transformations in an MDA Software Development Process. In Rensink, A., ed.: CTIT Technical Report TR-CTIT-03-27, Enschede, The Netherlands, University of Twente (2003) 121–126
14. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. 1st edn. Addison Wesley, Reading, Massachusetts, USA (2000)
15. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. In: Proceedings of the 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA, IEEE Computer Society (2003) 187–197
16. Almeida, J.P., Dijkman, R., van Sinderen, M., Pires, L.F.: On the Notion of Abstract Platform in MDA Development. In: The 8th International IEEE Enterprise Distributed Object Computing Conference, Monterey, California, USA, IEEE Computer Society (2004) 253–263
17. Tekinerdoğan, B., Bilir, S., Abatlevi, C.: Integrating Platform Selection Rules in the Model-Driven Architecture Approach. In Aßmann, U., ed.: Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden, Research Center for Integrational Software Engineering, Linköping University (2004) 184–200
18. Bézin, J., Devedžić, V., Djurić, D., Favreau, J., Gašević, D., Jouault, F.: An M3-Neutral infrastructure for bridging model engineering and ontology engineering. In: First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05), Geneva, Switzerland, Springer-Verlag (2005)
19. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Müthig, D., Paech, B., Wüst, J., Zettel, J.: Component-Based Product Line Engineering with UML. 1st edn. Addison Wesley, Reading, Massachusetts, USA (2001)
20. Pelechano, V., Pastor, O., Insfrán, E.: Automated code generation of dynamic specializations: an approach based on design patterns and formal techniques. *Data and Knowledge Engineering* **40** (2002) 315–353
21. Martinez, J.J., Salavert, I.R.: A Conceptual Model for Context-Aware Dynamic Architectures. In: Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03), Providence, Rhode Island, USA, IEEE Computer Society (2003) 138–143
22. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis(SM) Approach. Addison Wesley, Reading, Massachusetts, USA (1998)
23. Strang, T., Linnhoff-Popien, C., Frank, K.: CoOL: A Context Ontology Language to enable Contextual Interoperability. In: Proceedings of 4th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS2003), Paris, France, Springer-Verlag (2004) 236–247
24. Chen, H., Finin, T., Joshi, A.: An Ontology for Context-Aware Pervasive Computing Environments. *Knowledge Engineering Review* **18** (2004) 197–207 Special Issue on Ontologies for Distributed Systems.
25. Gu, T., Wang, X.H., Pung, H.K., Zhang, D.Q.: An Ontology-based Context Model in Intelligent Environments. In: Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS'04), San Diego, California, USA (2004) 270–275
26. Henriksen, K., Indulska, J.: A Software Engineering Framework for Context-Aware Pervasive Computing. In: Proceedings of the 2nd IEEE International Conference on Pervasive Computing and Communications (PerCom'04), Orlando, Florida, USA, IEEE Computer Society (2004) 77–86