

Explicit Platform Models for MDA

Dennis Wagelaar* and Viviane Jonckers

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

Abstract. The main drive for Model-Driven Architecture is that many software applications have to be deployed on a variety of platforms. The way MDA achieves this is by transforming a platform-independent model of the software to a platform-specific model, given a platform model. In current MDA approaches, the model transformations implicitly represent this platform model. Therefore, the number of different target platforms is limited to the number of supported model transformations. We propose a separate platform model, based on description logics, that can be used to automatically select and configure a number of reusable model transformations for a concrete platform. This platform model can be extended to describe the relevant platform information, including concrete platform instances as well as platform constraints for each model transformation. This separates the model transformation concern from the platform concern and, since the model transformations are no longer limited to targeting one platform, more platforms can be supported with the same set of transformations.

1 Introduction

The Model-Driven Architecture (MDA) allows for “separating the specification of the operation of a system from the details of the way that system uses the capabilities of its platform.” [1]. This enables the deployment of software applications on a variety of different platforms. The MDA pattern involves modelling the software in a platform-independent model (PIM). This PIM should then be transformed to a platform-specific model (PSM), given a platform model (PM). In current model transformation approaches for MDA [2], the model transformations themselves implicitly represent this platform model. As such, each platform requires one or more corresponding model transformations, which are specifically configured for that platform only. Because the platform concern is not separated from the model transformation concern, the number of supported target platforms is limited to the number of supported model transformations.

In practice, this means that only a relatively small number of general platforms can be targeted, e.g. Java, EJB [3] or C++. Targeting very specific platforms, e.g. Qtopia Palmtop Environment[4] or J2ME Mobile Information Device

* 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)

Profile 1.0 [5], is not feasible because of the maintenance overhead, even though such precise targeting can result in a better optimised PSM in terms of memory footprint, available features, etc. Especially in a world where constrained computing devices become more commonplace every day [6], getting the most out of such a platform is very important.

On the other hand, most model transformations are reusable over multiple platforms and it is only how they are configured that makes them applicable only to one specific platform. For example, one model transformation could target all Java 2 platforms by transforming UML “to-many” association ends 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 computer [7]. 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 separate platform model, which can be used to automatically select and configure a number of reusable model transformations for a concrete platform. This platform model is expressed in the Web Ontology Language (OWL) [8], which is an extensible language for describing ontologies. Furthermore, we use the OWL-DL variant, which corresponds to description logics [9], such that computational completeness can be guaranteed. This platform model forms a basis for describing platforms in general and can be extended to include the specific platform information that is relevant for a particular application domain. The model transformations can be augmented with a platform constraint that refers to the platform model. This way, the model transformations are no longer limited to one platform, but can instead be used for a well-defined class of platforms. An automatic reasoner, such as RACER [10] or Pellet [11], can be used to verify whether a concrete platform satisfies the platform constraints of a model transformation.

Section 2 explains in detail how platforms can be modelled. The definition of platform dependency constraints is discussed in section 3 and is illustrated by an example PIM and example model transformations. Section 4 explains how relevant model transformations are selected, based on their platform constraints and the concrete platform description. Section 5 discusses related work and section 6 concludes this paper.

2 Modelling Platforms

In order to reason about platforms and platform constraints, an ontology of platforms is used. Ontologies can serve as a common vocabulary for a domain [12]. The relationships between the ontology elements can be used to reason about elements based on that ontology, even if those elements aren’t related directly. A platform ontology allows one to base expressions about a platform on the vocabulary expressed by the ontology. By using a shared model of platforms, we can reason about the relationship between a platform description and a platform

constraint, even if the two do not have a direct relationship. An example platform constraint is that the Java 2 Collections framework needs to be present. An example of a platform description is a Sharp Zaurus hand-held computer. Since both the platform constraint and the platform description refer to the platform ontology to explain what the Java 2 Collections framework and the Zaurus hand-held computer are, one can derive whether the Zaurus hand-held computer platform satisfies the Java 2 Collections framework constraint.

2.1 A Platform Vocabulary

Before modelling any specific platforms or platform properties, a basic structure needs to be defined, into which platform extensions can be fitted. We will use a predefined ontology for describing computing context [13], which includes the platform. This ontology is in turn inspired by the User Agent Profile specification (UAProf) [14] and Composite Capability/Preference Profiles (CC/PP) [15], 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 is shown in Fig. 1.

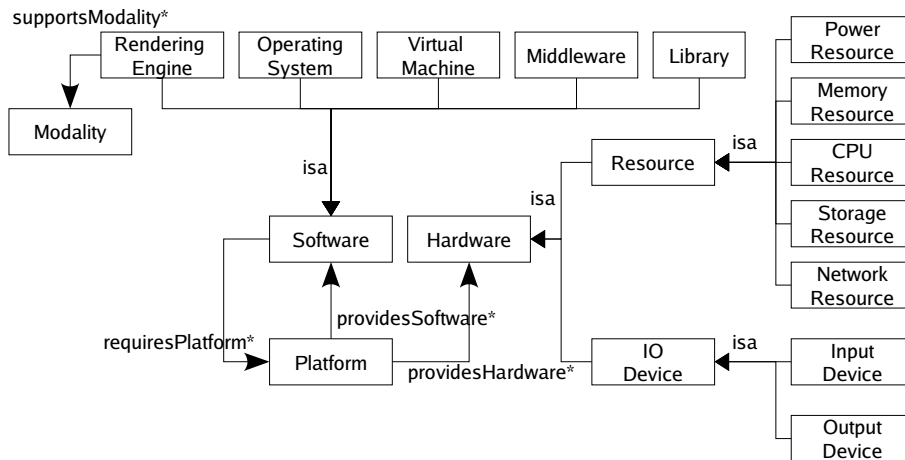


Fig. 1. Part of the context ontology for describing platforms

The platform concept in this ontology can provide software and hardware. A ‘*’ next to the relationship names 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 ontology can be extended for particular domains of platforms, such as Java virtual machines. Fig. 2 shows part of such an ontology. Above the line is a taxonomy of the Java virtual machines themselves. The “VirtualMachine” concept starts with “context:” to indicate it refers to the “VirtualMachine” concept from the main context ontology. The “JavaVM” can be subdivided in many different configurations. The “JDK” was the first Java configuration. “J2SE” refers to the virtual machines based on JDK 1.2 or up. “PersonalJava” is an early version of Java for mobile devices, which was later re-done as “J2ME”. “J2ME” offers two main configurations, “CDC” and “CLDC”, which are in turn refined by several sub-profiles. Each of these virtual machine classes implies a specific set of libraries and rendering engines. This is shown below the line: the “JDK” includes a simple “AWT” rendering engine, whereas “JDK1.1” already supports event-driven AWT, as does “CDC”. “J2SE” includes “Swing” in addition and “MIDP” has its own rendering engine, named “LCDUI”. Similarly, different versions of the `java.util` library are included in the “CLDC”, “JDK”, “CDC” and “J2SE” virtual machines.

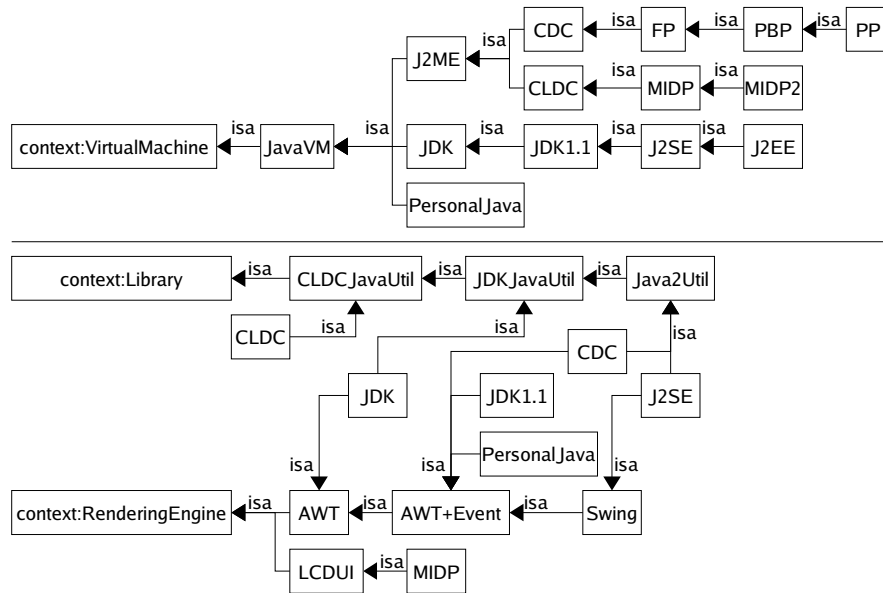


Fig. 2. Part of an ontology describing Java virtual machines

2.2 Modelling Concrete Platforms

Given the base context ontology and the extensions for the relevant domains, we can model concrete platforms as ontology instances. The Sharp Zaurus PDA, for example, has a J2ME PP virtual machine. The ontology extension that describes this is shown in Fig. 3.

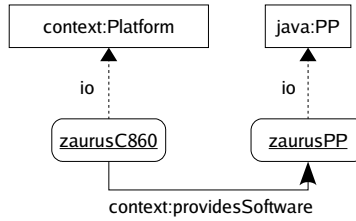


Fig. 3. Partial platform description for the Sharp Zaurus PDA

The concepts “Platform” and “PP” are taken from the context and 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. Finally, the “zaurusC860” platform has a “provides-Software” relationship with the “zaurusPP” Java Personal Profile virtual machine.

3 Modelling Platform Dependencies

The platform dependencies for a particular model transformation can be modelled by defining new, *completely specified* concepts. Such concepts have *necessary-and-sufficient* constraints in addition to any *necessary* constraints. A *necessary* constraint is depicted by the “isa” relationship: whereas it is *necessary* that each “JDK” instance is also an instance of “JavaVM”, being a “JavaVM” instance is not *sufficient* for also being a “JDK” instance (see Fig. 2). We will use the notation for describing conditions as used in the Protégé ontology modelling tool [16]. A constraint that requires a platform with an “AWT” rendering engine can be defined as a concept “JavaAWTPlatform”, which is a sub-concept of “Platform” (*necessary*) and provides an “AWT” rendering engine (*necessary-and-sufficient*):

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

Whenever a “Platform” instance fulfils the condition of providing an “AWT” rendering engine, it can be classified as an instance of “JavaAWTPlatform”. This 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. 3 classifies as an instance of “JavaAWTPlatform”, since “zaurusPP” is an instance of “PP”, which is a sub-concept of “AWT”.

3.1 Example PIM

Fig. 4 shows the UML class diagram of part of the PIM for a simple instant messaging client. The instant messenger client is able send and 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 ErrorReporter interface: it reports raised exceptions either on the command line or on a Network that implements ErrorReporter (e.g. a Loopback network). The design is split up in a model, edit, view and networking part, each in their own package. Concrete view and network types are not shown in the class diagram and will not be considered for the purpose of our example.

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 can be defined to translate each of these elements to one or more elements that are available in the target programming environment. Examples of some of these model transformations will be discussed below. The ATL transformation language [17], which has a simple, rule-based syntax, will be used to express these examples.

3.2 Example Model Transformations

A common way for transforming UML 1.5 [18] associations to corresponding attributes in Java is to 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 association²:

```
rule AssocToSingleAttr {
  from s : INMODEL!AssociationEnd (
    s.isNavigable and
    s.multiplicity.range->select(r|r.upper<>1)->isEmpty())
  to t : OUTMODEL!Attribute mapsTo s (
    name <- s.name,
    owner <- s.association.connection->select(x|x<>s)->first().participant,
```

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

² 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.

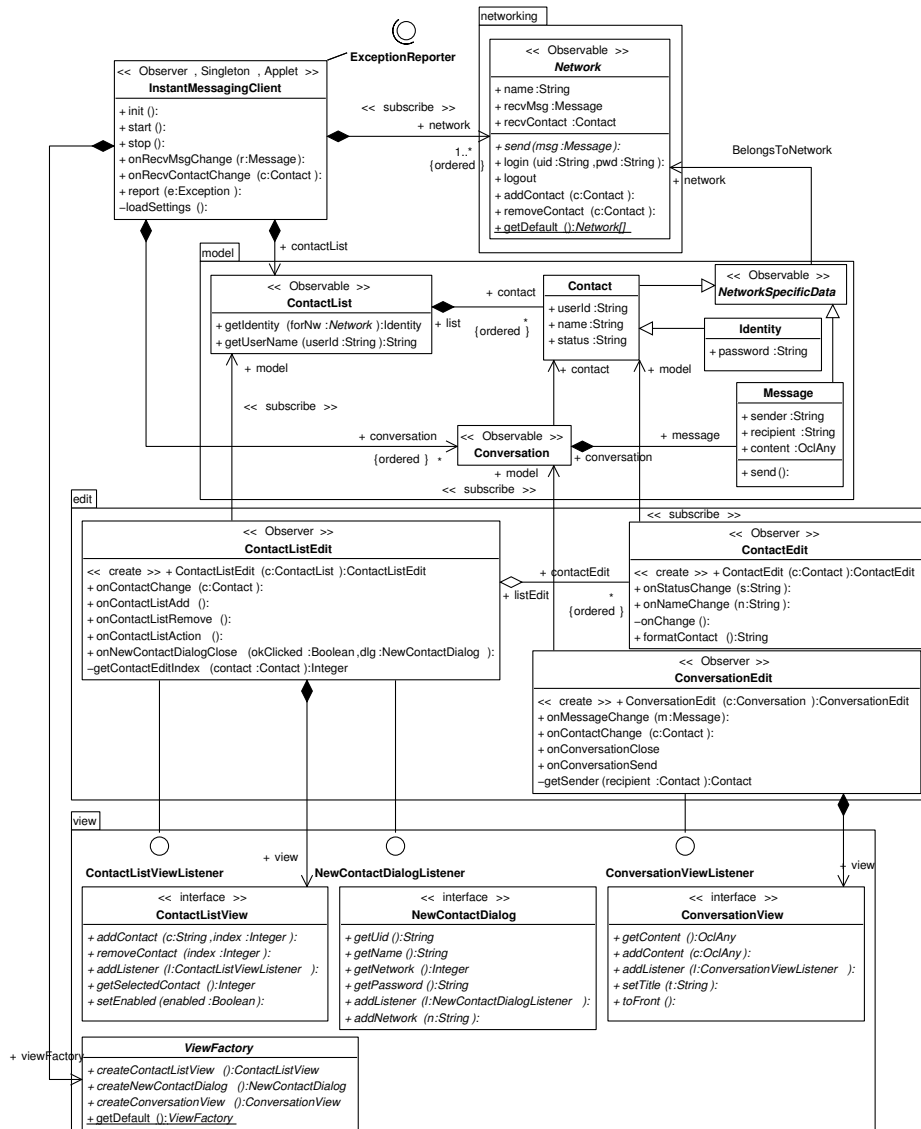


Fig. 4. Example PIM class diagram for a simple instant messaging client

```

    type <- s.participant ,
    visibility <- s.visibility ,
    ownerScope <- s.targetScope ,
    changeability <- s.changeability)
}

rule AssocToArrayList {
  from s : INMODEL!AssociationEnd(
    s.isNavigable and
    not s.multiplicity.range->select(r|r.upper<>1)->isEmpty())
  using { list : INMODEL!Interface =
    INMODEL!Interface.allInstances()->select(c|c.name='List')->first(); }
  to t : OUTMODEL!Attribute mapsTo s (
    name <- s.name ,
    owner <- s.association.connection->select(x|x<>s)->first().participant ,
    type <- list ,
    visibility <- s.visibility ,
    ownerScope <- s.targetScope ,
    changeability <- s.changeability ,
    initialValue <- value) ,
  value : OUTMODEL!Expression (
    language <- 'java' ,
    body <- 'new java.util.ArrayList();')
}

```

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. The second rule has a **using** clause, which locates the Java List interface. This List interface is then used as the type of the attribute that is created. The `ArrayList` class is used for initial value of this attribute.

The **AssocToSingleAttr** transformation rule does not use any Java-related elements, and has no platform dependencies. The **AssocToArrayList** rule uses the Java 2 Collections framework, which is part of the “Java2Util” library from Fig. 2. This corresponds to the following constraint:

$$\begin{aligned}
 \text{Java2UtilPlatform} &\sqsubseteq \text{context} : \text{Platform} \\
 &\equiv \exists \text{context} : \text{providesSoftware } \text{java} : \text{Java2Util}
 \end{aligned}$$

An alternative for the **AssocToArrayList** rule could use the `java.util.Vector` class to implement the one-to-many association:

```

rule AssocToVector {
  from s : INMODEL!AssociationEnd(
    s.isNavigable and
    not s.multiplicity.range->select(r|r.upper<>1)->isEmpty())
  using { vector : INMODEL!Class =
    INMODEL!Class.allInstances()->select(c|c.name='Vector')->first(); }
  to t : OUTMODEL!Attribute mapsTo s (
    name <- s.name ,
    owner <- s.association.connection->select(x|x<>s)->first().participant ,
    type <- vector ,
    visibility <- s.visibility ,
    ownerScope <- s.targetScope ,
    changeability <- s.changeability ,
    initialValue <- value) ,
}

```

```

value : OUTMODEL!Expression (
  language <- 'java',
  body <- 'new java.util.Vector();')
}

```

Because the Java Vector class is already available in the “CLDCJavaUtil” library from Fig. 2, the platform constraint can be relaxed to only requiring a “CLDCJavaUtil” library:

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

4 Selecting Model Transformations

In order to select which model transformations need to be applied, the transformations are grouped into sets of alternatives that represent the same functionality. This grouping can be (partially) automated, based on a heuristic that checks the input specification of the transformation rules. If certain transformation rules have the same input specification, then they are considered alternatives. The transformation rules `AssocToArrayList` and `AssocToVector`, 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 can be adapted manually afterwards. The grouping information is reusable over multiple PIMs and PMs: it only has to be re-computed if the set of transformation rules changes. An example grouping for the model transformations needed for our example PIM is shown in Table 1.

<code>AssocToArrayList</code> <code>AssocToVector</code>
<code>Observer</code> <code>JavaObserver</code>
<code>Accessors</code> <code>Java2Accessors</code>
<code>Applet</code> <code>MIDlet</code>
<code>Singleton</code>
<code>DataTypes</code> <code>Java2DataTypes</code>

Table 1. Example model transformations grouping

The `AssocToArrayList` and `AssocToVector` transformations have already been discussed and form one group. The `Observer` and `JavaObserver` transformations both implement the “Observer”, “Observable” and “subscribe” stereotypes. The first transformation requires no Java API, while the second uses the Java 1.0 `java.util.Observer` interface and the `java.util.Observable` class. The `Accessors` transformation creates accessor operations (getters and setters)

for each public attribute. **Java2Accessors** does the same, but uses the Java 2 Collections data types. The **Applet** transformation transforms all classes with the “Applet” stereotype into Java applets, whereas the **MIDlet** transformation transforms the same classes into J2ME MIDlets. The **Singleton** transformation adds the singleton infrastructure to each class with the “Singleton” stereotype. Finally, the **DataTypes** and the **Java2DataTypes** transformations translate the OCL data types into Java data types and Java 2 Collections data types respectively.

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 (semi-)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 **Java2Accessors** transformation states that it creates new operations. If another model transformation, **JavaObserver**, adapts all setter accessor operations for each **Observable**, then its input specification could match elements generated by **Java2Accessors**. Hence, **Java2Accessors** has to be placed before **JavaObserver**. 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. The sorting information is, again, reusable over multiple PIMs and PMs: it only has to be re-computed if the set of transformation rules changes. The sorted list of transformation groups is shown in Table 2.

AssocToArrayList AssocToVector
Accessors Java2Accessors
Observer JavaObserver
Applet MIDlet
Singleton
DataTypes Java2DataTypes

Table 2. Example model transformations grouped and sorted

From each group, at most one model transformation is selected. This selection is based on platform relevance. The platform 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 platform-specific³. Consider the following platform constraint:

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

³ This criterion is based on the “Do the Most Specific” conflict resolution strategy in Forward Chaining reasoners such as OPS5 [19] and successors.

Compared to the “Java2UtilPlatform” constraint, this constraint requires the “CLDCJavaUtil” library instead of the “Java2Util” library. Since “Java2Util” subsumes “CLDCJavaUtil” (see Fig. 2), “Java2UtilPlatform” also subsumes “CLDCUtilPlatform”. Reasoner engines, such as RACER, can automatically determine a taxonomy of constraints using such inference rules. Note that this taxonomy can be determined without knowledge of any concrete platform (i.e. the platform instances). As such, this taxonomy can be pre-computed and only needs re-computing if the set of constraints changes.

One can imagine that there are cases in which one cannot determine whether a constraint concept subsumes another constraint concept. Consider the following constraint:

$$\begin{aligned}
 \text{AWTUtilPlatform} &\sqsubseteq \text{context} : \text{Platform} \\
 &\equiv \exists \text{context} : \text{providesSoftware } \text{java} : \text{CLDCJavaUtil} \wedge \\
 &\quad \exists \text{context} : \text{providesSoftware } \text{java} : \text{AWT}
 \end{aligned}$$

When comparing this constraint to “Java2UtilPlatform”, no conclusion can be made on which is more specific. While “CLDCJavaUtil” is subsumed by “Java2Util”, nothing can be said about “AWT”, since no comparable rule occurs within the “Java2UtilPlatform” constraint. It is very unlikely that functionally equivalent model transformations (i.e. alternatives from one group) will have orthogonal (parts of) constraints (e.g. one constraint requires “AWT” whereas another requires nothing of the sort). However, to cope with this case, one can manually order a group of model transformations, such that the first most-specific is chosen.

When selecting the model transformations for a specific platform, (1) the local constraint taxonomy is taken for each group of alternative transformations and (2) pruned such that all constraints that do not hold are removed. Note that, if no transformations are left for a particular group after this step, no PSM can be generated for the given platform. Then, (3) the first model transformation for which the constraint forms a leaf in the taxonomy tree is chosen. All of the steps 1-3 are of linear complexity⁴, so the selection mechanism in its entirety also performs in linear time.

The list of chosen transformations for the example platform from Fig. 3 is shown in Table 3.

The `AssocToArrayList` transformation was chosen over the `AssocToVector` transformation, because it requires a “Java2Util” library instead of a “CLDCJavaUtil” library. For the same reason, the `Java2Accessors` transformation is chosen over `Accessors`. The `JavaObserver` transformation is chosen over the `Observer` transformation, because it requires a “JDKUtil” library instead of just any “JavaVM”. `Applet` is chosen over `MIDlet` because the constraint of the latter did not hold (a `MIDlet` requires `J2ME MIDP`). Finally, `Java2DataTypes` is chosen over `DataTypes`, because it again requires a “Java2Util” library instead of a “CLDCJavaUtil” library.

⁴ the set of transformations and the set of constraints are constant at this time

AssocToArrayList
Java2Accessors
JavaObserver
Applet
Singleton
Java2DataTypes

Table 3. Example selection of model transformations

5 Related Work

Model transformations are subject to similar configuration management issues as regular software components [20]. Transformation dependencies can be made explicit through their input and output specifications. Limited versioning support is provided by the platform constraints: one can discriminate on platform-specificity of different versions of a transformation.

In Generative Programming [21] and Step-Wise Refinement [22], *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 by model transformations, which are chosen based on platform 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 also discussed in [23]. They introduce *abstract platforms*, which describe 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 platform models are meant to decouple model transformations from concrete platforms.

In [24], 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 scenario [6] 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 [25], an infrastructure for combining UML/MOF models and ontologies is introduced. Such an infrastructure can be useful for a better integration of platform constraints into model transformation languages that are based on MOF.

6 Conclusion and Future Work

This paper has introduced a platform modelling framework that can describe platform constraints as well as instances of concrete platforms. By separating the platform concern from the model transformation configuration and moving it to an explicit platform model, the model transformations can be reused over several platforms. Our framework can automatically select a number of applicable model transformations for a specific platform. This is done by matching platform constraints for each model transformation against a concrete platform description. Both of these are based on a common ontology, described in OWL-DL, such that an automatic reasoner can determine whether the platform matches the constraint. In this way, more than one concrete platform can be supported with the same set of model transformations.

Note that the reusability of the individual model transformations remains the same. Only the configurations of model transformations, which are far less reusable than an individual transformation, are now automatically derived by means of an explicit platform model. As such, the separation of the platform concern from the model transformation concern is not complete, since each model transformation must include the local platform information that it is supposed to add to the PSM.

The chosen ontology for modelling platforms may not be general enough for all cases and it may also not be specific enough in some cases. Experience will have to show how far we can go with the current ontology. However, we expect that our mechanism can also be applied to different ontologies, since it only requires that the constraints are expressed in description logic.

For the selection mechanism, all the potentially expensive calculations are done in advance (i.e. transformation grouping and sorting and calculating the subsumption taxonomy of constraints). Only if the set of transformations or the set of constraints on those transformations changes, these calculations need to be redone. When transforming a PIM to a PSM, using a concrete platform model, three steps are performed to select which model transformations need to be applied. Each of these steps are of linear complexity, such that the mechanism in its entirety also performs in linear time. As such, the proposed mechanism should scale sufficiently.

Using the current ordered lists of model transformation groups, only a limited set of model transformation dependencies can be expressed. The example used in this paper shows a group containing the `AssocToArrayList` and `AssocToVector` transformations and another group containing `Accessors` and `Java2Accessors`. While we can express that the second group depends on the first, we cannot express that, for the sake of type consistency, `AssocToArrayList` may only be applied in combination with `Java2Accessors`. Recent work on feature modelling [26,27] and product families [28] provides promising approaches for modelling such complex dependencies and do automatic reasoning on them. We will also investigate if we can map these feature modelling approaches to description logic in order to integrate feature modelling with our platform modelling approach.

Acknowledgement

The author would like to thank Ragnhild Van Der Straeten, Bruno De Fraine and Wim Vanderperren for reviewing a draft of this paper. Furthermore, the author would like to thank the CoDAMoS project user committee for discussing their ideas for the Instant Messaging scenario, which were useful for the example PIM in this paper.

References

1. Miller, J., Mukerji, J.: MDA Guide. Object Management Group, Inc. (2003) Version 1.0.1 (omg/03-06-01).
2. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture. (2003)
3. DeMichiel, L., Ümit Yalçınalp, L., Krishnan, S.: Enterprise JavaBeans™ Specification. Sun Microsystems, Inc. (2001) Version 2.0.
4. Trolltech: Qtopia application platform for embedded Linux. (2005) [Online] <http://www.trolltech.com/products/qtopia/>.
5. Sun Microsystems, Inc.: Java 2 Micro Edition website. (2005) [Online] <http://java.sun.com/j2me/>.
6. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.C.: Scenarios for Ambient Intelligence in 2010. IST Advisory Group (ISTAG). (2001) [Online] <ftp://ftp.cordis.lu/pub/ist/docs/istagscenarios2010.pdf>.
7. Sun Microsystems, Inc.: Java 2 Standard Edition website. (2005) [Online] <http://java.sun.com/j2se/>.
8. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. World Wide Web Consortium. (2004) W3C Recommendation 10 February 2004.
9. 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)
10. Möller, R., Haarslev, V.: Description Logics for the Semantic Web: Racer as a Basis for Building Agent Systems. *Künstliche Intelligenz* (2003) 10–15
11. Parsia, B., Sirin, E., Grove, M., Alford, R.: Pellet website. Mindswap. (2005) [Online] <http://www.mindswap.org/2003/pellet/>.
12. Gruber, T.R.: A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition* **5** (1993) 199–220
13. 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
14. Open Mobile Alliance: User Agent Profile 2.0 Specification. (2003) Version 20-May-2003.
15. 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)
16. Stanford Medical Informatics, Stanford University School of Medicine Stanford, CA, USA: Protégé Project website. (2005) [Online] <http://protege.stanford.edu/>.

17. 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)
18. Object Management Group, Inc.: Unified Modeling Language Specification. (2003) Version 1.5 (formal/03-03-01).
19. Brownston, L., Farrell, R., Kant, E., Martin, N.: Programming expert systems in OPS5: an introduction to rule-based programming. Addison Wesley, Reading, Massachusetts, USA (1985)
20. Larsson, M.: Applying Configuration Management Techniques to Component-Based Systems. Licentiate thesis, Department of Information Technology, Uppsala University, Uppsala, Sweden (2000) Also published as report MRTC 00/24 at Mälardalens högskola.
21. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. 1st edn. Addison Wesley, Reading, Massachusetts, USA (2000)
22. 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
23. 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
24. 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
25. Bézivin, 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)
26. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Proceedings of the 17th Conference on Advanced Information System Engineering (CAiSE'05), Porto, Portugal (2004)
27. Klint, P., van der Storm, T.: Reflections on Feature Oriented Software Engineering. In: OOPSLA Workshop on Managing Variabilities Consistently in Design and Code (MVCDC 2004), Vancouver, Canada (2004)
28. Liu, J., Batory, D.: Automatic Remodularization and Optimized Synthesis of Product-Families. In: Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE 2004), Vancouver, Canada, Springer-Verlag (2004) 379–395