Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 151–161

www.elsevier.com/locate/entcs

Refinement via Consistency Checking in MDA

Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack[1]

Department of Computer Science
University of York
York, UK

Abstract

Refinement is a key practice in the Model-Driven Architecture initiative of the Object Modelling Group. However, the practice is loosely defined, overloaded, and open to misinterpretation. In this paper, we outline ongoing work on providing a precise definition for refinement via consistency checking, not only in the context of MDA, but more generally for model-driven development in a variety of domains.

Keywords: Model-Driven Architecture, Consistency, Traceability, Refinement.

1 Introduction

Model-Driven Development (MDD) attempts to raise the level of abstraction by which software and systems engineers carry out their tasks. This is done by emphasising the use of *models* - i.e., abstractions - of the artifacts that are developed during the engineering process. Models are representations of phenomena of interest, and in general are usually easier to modify, update, and manipulate than the artifact or artifacts that are being represented. Models are expressed using a suitable modelling language; UML is a widely used and recognised standard in MDD [12].

[1] Email: paige@cs.york.ac.uk, dkolovos@cs.york.ac.uk, fiona@cs.york.ac.uk

doi:10.1016/j.entcs.2005.04.029

MDD is not a development method or process; it can be implemented in a number of ways, e.g., via Extreme Programming, the Rational Unified Process, the B-Method, or a refinement calculus. The key element in MDD is the construction and transformation of models that are fit for the purposes of the development project. The languages and processes used in construction and transformation will vary from project to project.

The Model-Driven Architecture (MDA) [1] is an initiative of the Object Modelling Group (OMG) [2], aimed at providing a standard approach for MDD. While MDD does not prescribe the use of specific technologies or tools, nor a specific process or sequence of steps to follow, MDA requires the use of a standard modelling language – UML - and meta-steps that should be followed in the development of models and systems.

Perhaps surprisingly, refinement is a key concept in MDA. This is surprising for two reasons. First, the languages prescribed in MDA are UML-based and generally do not provide the level of formality one might expect in a refinement-based method. Second, the languages used in MDA are *multi-view*, and as such it is difficult to achieve seamless refinement-based development with them.

Even though refinement is a (and perhaps even *the*) key concept in MDA, the term is loosely defined, and open to misinterpretation. Part of this is because of the relative semi-formality present in the modelling languages used in MDA; but it is also because of the relative immaturity in MDA at the present time.

In this paper, we report on ongoing work, in the context of MDD and MDA, attempting to capture rules for obtaining consistent models. We propose that these consistency rules can be used to provide a formal definition of refinement in MDA. We start with a brief overview of MDA, and give its (informal) notion of refinement, as well as its process. We then describe model consistency (in the context of MDD and MDA) and then suggest how consistency can be used to achieve refinement. We end by a discussion of ongoing and future work. Parts of this work have been carried out in the context of the EU Integrated Project "Modelware" [11].

2 Background and Related Work

2.1 Overview of MDA

There are a number of proposed benefits that can be obtained by MDD, particularly increased portability and productivity. The Model-Driven Architecture attempts to standardize elements of MDD by defining a common language for specifying models, and a common sequence of steps for generating models.

Thus, MDA attempts to achieve the benefits of MDD while also improving interoperability (different projects can share models) and making it easier to integrate data and applications by standardizing the system development process.

One of the key ideas in MDA is the notion of *platform*, and correspondingly, platform independence. The MDA guide defines a platform as [1]: "...a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns..." Platform independence means that a system must be modelled in such a way that the model is independent of any underlying platform of interest. This is vague and encompasses platforms like CORBA, .NET, and J2EE, though certainly operating systems and programming languages can also be included. Such abstract models are termed *platform-independent models (PIMs)*, as opposed to models that include platform details, i.e., *platform-specific models (PSMs).*

The MDA process involves successive refinement. Refinement steps generally are of two types.

- Transformations (also called mappings), which generate a new kind of model, e.g., transforming a PIM into a PSM, or a PSM into executable code. Such transformations in general add detail to the model. By analogy to refinement calculi, these transformations reduce nondeterminism by making design decisions, e.g., how to represent data, how to implement messaging.
- Internal refinements, which are semantics-preserving transformations applied to a model, and which produce the same kind of model, e.g., refining a PIM into a new PIM.

The overall MDA process is illustrated in Fig. 1.

In general, feedback between stages (e.g., between PSM and PIM) will be obtained and used. This may make it possible - though perhaps not desirable - to keep PIM and PSM consistent, e.g., via reverse engineering techniques.

The issue of what constitutes a valid refinement step is an interesting one. The idea in MDA is that predefined transformations (written in a standard transformation language, QVT [6]) will be applied in order to go from PIM to PSM, or from PSM to code. Custom (often hand-written) refinement steps will be used in order to refine within a particular type of model, e.g., from PIM to PIM. The latter type of transformation is usually dependent on the application context. Thus, it is assumed that in general many of the transformations that are being applied have been previously validated by an MDA expert, and thus are safe to apply - providing that the rules for applying the transformation are obeyed[2]. However, this does not help in validating custom, hand-written

[2] For example, a transformation that replaces multiple generalization in UML with single

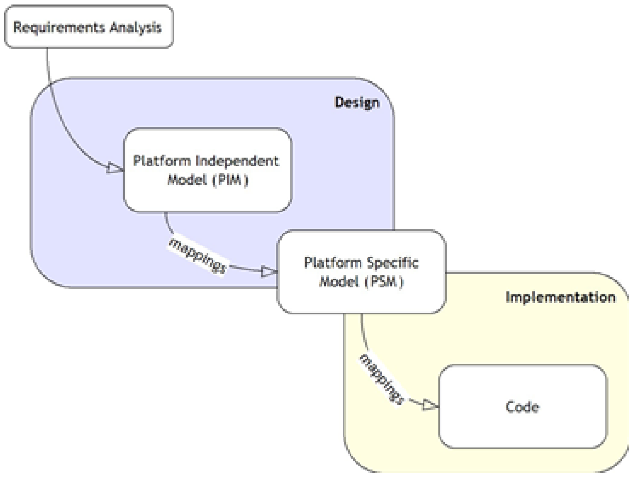
Requirements Analysis
Design
Platform Independent Model (PIM)
mappings
Platform Specific Model (PSM)
Implementation
mappings
Code

Fig. 1. The MDA successive refinement/transformation process

transformations, nor does it help the MDA expert who must validate the reusable transformations that are to be stored in an MDA library.

2.2 Refinement in MDA

The MDA guide is vague in its definition of MDA and the notion of refinement. The guide defines MDA in terms of PIM, PSM, and additional models such as domain models. Refinement is defined informally as a process of transforming MDA models (e.g., PIM to PSM, PSM to code, PIM to PIM). The MDA guide distinguishes PIM and PSM as models at different levels of abstraction, e.g., a PIM is at a higher level of abstraction than a PSM. However, years of research on refinement calculi and programming methodology, particularly on wide-spectrum languages, suggest that distinctions such as this are not helpful: it is more productive to think in terms of specifications (or, in MDA terms, models) that have different properties. For example, in Hehner's predicative programming [13], programs are a special kind of specification: they are implementable and immediately executable on a machine. Similarly, in Morgan's refinement calculus [14], specifications are a special kind of program: they are not always executable, but one can test for feasibility, and they are written in a unified language.

To formally define refinement in MDA, there are four alternatives. One

generalization and delegation might include a precondition that (a) an instance of multiple generalization exists in the PIM, (b) the classes being generalized are not all abstract, and (c) the target language supports the particular instantiation of the delegation pattern that we are using.

could translate the core languages used in MDA i.e., UML, or a subset of UML into a formal language such as Z, B, or specification statements. Work has been carried out on expressing such translations, but it all suffers from limitations, e.g., incompleteness, difficulties in achieving consistency, etc. A second alternative is to promote a formal definition of refinement e.g., weakest preconditions and express it in MDA terms, e.g., in UML. It is debatable whether UML is well suited to expressing formal definitions of refinement.

The third alternative, which appears in the Catalysis method [15], is to model refinement retrieve relations using UML associations. This is attractive and lightweight, though incomplete since it provides the necessary traceability features of refinement but not the specific constraints needed to express retrieve relations; the latter requires some form of predicate logic.

The fourth alternative is to define refinement in terms of model consistency. We explore this in the next section.

3 A Consistency-Based Definition of MDA Refinement

There are two essential parts to MDA: languages for expressing models, and transformations applied to models. The languages used in MDA are generally UML-based, and as such are built atop the Meta Object Facility (MOF) [10], which is a core language for defining languages. Thus, the transformations used in MDA are defined in terms of MOF and its core constructs, e.g., classes, objects, features.

The rigorous way to define refinement in MDA would be to provide a formal semantics for MOF and then formally define refinement in terms of this semantics. It is likely that this would be difficult to use for carrying out practical refinements and refinement steps; moreover, providing a generic formal semantics for MOF is challenging. Furthermore, using this generic formal semantics to carry out refinement in MOF-based languages such as UML may not be straightforward, and would require the developer to make use of not only the formal semantics, but MOF, in order to carry out refinement steps.

The difficulties associated with this rigorous approach to defining refinement in MDA are typical of attempts to attach formal analysis techniques to semiformal languages after the languages have been defined. The difficulties are perhaps more pronounced in MDA because of the layered nature of the languages used, i.e., UML defined in terms of MOF.

An alternative, perhaps more lightweight approach to defining refinement in MDA is to define it in terms of consistent models. This is the approach we are taking in the Modelware project, and we now outline the general technique, starting with some definitions.

3.1 Formal Definitions for Model Consistency

A *model* is a representation of phenomena of interest. The representation can be concrete (e.g., written down on paper, drawn on a whiteboard, on a screen) or can be left unstated (e.g., a 'mental model').

A *view* of a system model is an explicit description of the system from one perspective of interest. Perspectives of interest are language and model dependent, but typical perspectives include: behavioural perspectives, architectural perspectives, user/stakeholder perspectives. A view often captures a subset of a system model, but this is not always the case. For example, in MDA, an executable implementation of a PSM is a view of a system model, but it is not a subset, as it can be executed in full. In UML, typical views include structural views (provided by class diagrams) and behavioural views (provided by statecharts).

A *diagram* can be used to express one or more views of a model; non-graphical languages can also be used to capture views. A diagram is expressed in a language. The language itself consists of a syntax and a metamodel, which defines the well- formedness constraints for the language. For example, the metamodel for UML class diagrams will contain a constraint stating that all classes in a diagram must have unique names.

It is important for a model to be consistent. We define consistency rigorously in the sequel, but informally, a consistent model is one that has at least one implementation. Inconsistent models may include errors or omissions. If the model is multi-view (e.g., consisting of class and communication diagrams in UML) then the diagrams used may contradict each other, e.g., a message that is sent in a sequence diagram cannot actually be sent according to the class diagram.

We now define model consistency formally. Let L be a language, consisting of a notation N and a metamodel MM, the latter of which is a set of rules. The metamodel for UML has its rules specified in the Object Constraint Language (OCL). For any model m we say that m is in the language (written $m \in L$) if the following two conditions hold.

(i) m **sat** N, i.e., m is syntactically valid according to the abstract or concrete syntax rules for the language L.

(ii) For each metamodel rule r, m **sat** r, i.e., m satisfies each well-formedness constraint for language L.

We can use the same definition for model consistency; that is, if $m \in L$ then m is consistent. This requires two simple extensions, as follows. The first is to clarify that the language L supports the definition of multiple views, i.e., that a model m can be expressed (syntactically) using two or more diagrams, and that (semantically) these diagrams can overlap. For example, UML possesses class diagrams and communication diagrams, and (semantically) these diagrams can overlap. We express this by being more precise about the definition of notation N. We say that N is a multi-view notation, i.e., that $N = \{V1, V2, \ldots, Vn\}$ where each Vi is a diagram with its own syntax. It is conceivable - even likely - that each Vi will induce an equivalence class of metamodel rules pertaining to the view presented by the diagram; we can express the metamodel as follows: $MM = \{R1, R2, ..., Rn, R\_all\}$, where Ri is the set of rules pertaining to diagram Vi, and R_all is the set of rules that pertain to two or more diagrams.

R_all is the set of multi-view consistency rules. For existing languages R_all may not be a complete set of rules, and thus we may want to add new ones. To extend the ruleset, we write:

$$MM = \{R1, R2, ..., Rn\} \cup R\_all \cup R\_new$$

where R_new is a set of new consistency rules; whether they are domain-specific or not is irrelevant to the consistency checking process.

3.2 Types of Consistency Rules

In general, the rules described in the previous section are of one of two types: structural rules (related to the structural properties of the diagrams used to represent the model) and state-based rules (related to the state of the diagrams representing the model). The latter are sometimes referred to as behavioural rules.

Structural rules are generally not difficult to check, and usually require checking that one or more diagrams satisfy the relevant rules in the metamodel. Consider a language consisting of class and sequence diagrams (e.g., a subset of UML). The rule set for this language will consist of the metamodel subsets for class and sequence diagrams, as well as a set of rules like the following.

- All the classes appear with at least one instance in at least one collaboration diagram of the model
- All methods and attributes used in the sequence diagram correspond to existing methods in the class diagram; there must be signature matching.
- If an instance of class A invokes a method of class B in the sequence diagram, this invocation appears in the class diagram via a dependency association.
- All public methods in the class diagram appear in at least one sequence diagram (completeness) a public method that does not appear in a sequence

diagram could result in a rule with action "recommend that this method change to private".

- Instances participating in a sequence diagram are not of abstract or interface type.

We can express rules like these (and others) in OCL so that OCL evaluation can be carried out. This process is quite straightforward and simple to implement. For example the OCL fragment below implements the first rule of the above list.

```
package Foundation::Core
context Class

     -- Check that all classes appear in at least one
     -- collaboration diagram
     -- Step 1: Select all the collaboration diagrams of the model
     -- Step 2: Check that there exists at least one instance
     --         in at least one collaboration diagram which whose
     --         classifiers include the current class

inv appearInCollaboration:
self.model.allContents()->select(me|me.oclIsTypeOf(Collaboration))->
      exists(c|c.oclAsType(Collaboration).allContents()->
      exists(o|o.oclAsType(Instance).classifier->includes(self)))

endpackage
```

State-based rules are much more challenging to check. Behavioural diagrams (e.g., communication diagrams in UML) imply sequences of actions and events that change the state of the participating elements. Therefore, the model might be structurally consistent but its state - the state of its elements - might deviate from a desirable one. From this point of view, two types of possible inconsistencies can be identified:

- After an action has taken place in a behavioural diagram, it must be checked that all other views (i.e., the overall system) are left in a consistent state that is, the invariants of all the elements hold.
- There might be preconditions (guards) for actions that contradict the invariants of the participating model elements. Therefore, under normal circumstances, those actions will never take place.

3.3 Consistency in MDA

In MDA, structural and state-based rules are insufficient to fully capture model consistency, since the overall system model is split into several parts, e.g., PIM and PSM. Thus, the rules discussed in the previous section must be applied to check for consistency. As well, at least two additional kinds of consistency must be checked.

- Consistency between the PIMs and PSMs: This type of consistency ensures that information is not lost or misinterpreted during the transition from platform independent to platform specific models. Within this category, the rules discussed earlier can be applied.

- Consistency of PSMs with the domain: Each domain (e.g., information systems, database design) has domain-specific rules that should not be broken. Since these rules might not be directly imposed by the modelling language syntax, extra mechanisms i.e., stereotypes or metamodel refinements must be applied in order to ensure their application. Thus, for each different application domain, a set of domain-specific rules must be checked.

PIM-PSM consistency can be checked in several ways. The most flexible approach is to check consistency via analysis, in much the same way as discussed in Section 3.2. In other words, cross-model consistency rules are captured (e.g., using OCL). These rules are more complex than the previous examples since they are cross-model (i.e., are specified in terms of both PIM and PSM concepts). This typically requires an integrated PIM and PSM metamodel in order to capture the rules.

The same approach can be used for PSM-domain consistency: a domain model (e.g., for databases) has a metamodel and a syntax and thus cross-model consistency rules can be captured in the same way as for PIM-PSM consistency. This is additional evidence that the distinction between PIM, PSM, and other models in MDA is not always particularly useful.

3.4 MDA Refinement is Model Consistency

We have described the MDA process as one of successive refinement, producing different kinds of models (e.g., PIM, PSM, domain). We have also suggested that the distinctions between these different types of models is not always particularly useful. The key idea is that within the MDA process, models are defined and modified and new models that are produced must "refine" previously constructed models. We suggest that a sensible, lightweight definition for refinement in MDA is exactly model consistency.

Definition 3.1 [MDA Refinement]: In the context of an MDA development, a model A is refined by a model B iff B is consistent with A, i.e., A and B obey the following rules.

(i) Both A and B obey the metamodel well-formedness rules for their respective languages.

(ii) Both A and B obey any multi-view consistency rules (e.g., A may be a model constructed from a class and sequence diagrams; these must be internally consistent).

(iii) A and B must obey any cross-model consistency rules relevant to their context. For example, if A is a PIM and B a PSM, then all PIM-PSM consistency rules relevant to the transformation of A into B must be satisfied. Similarly, if A is a PSM and B a domain model, then all PSM-domain consistency rules must be satisfied.

This definition has advantages over some of the alternatives for defining MDA refinement that were discussed earlier.

- It is lighweight and does not require the use of formal techniques that, while ideal and suitable for the problem, will likely be unpalatable to MDA developers.
- It is modular and extensible: different application domains and modelling languages can define their own notion of refinement while following the same underlying process and set of principles inherent in MDA. It is also compatible with the transformational nature of MDA: consistency rules, and hence refinements, are reusable across different domains.
- It can be implemented using tools that should be familiar and usable by MDA developers (discussed in Section 3.5)

The disadvantages of this approach generally relate to soundness and completeness: while the above definition is precise, it is not formal and thus it is not possible to check that the definition is sound and complete (i.e., all legitimate MDA refinements can be expressed in terms of it). Moreover, it does require substantial effort to build up a 'library' of valid refinements, especially as new application domains are used.

3.5 Implementing MDA Refinement

Implementing the above definition of MDA refinement is generally straightforward, though it does require expertise in metamodelling. The approach that we are taking in the Modelware project is to implement the consistency rules (including cross-model consistency rules) using an OCL engine. This OCL engine provides an OCL parser, type checker, and rule checker which simulates rules against models. The models themselves are written in a MOF-compatible language (i.e., a profile of UML). The rule checking process is fully automatic, and if a rule fails the feedback is provided in terms that the developer can understand. It remains to be seen if this approach will be considered usable and useful by MDA developers.

4 Conclusions

We have reported on ongoing work, in the context of an integrated EU project, on model consistency and formulating a precise definition of MDA refinement. We have attempted to equate the two, in order to provide a lightweight definition of MDA refinement that will appeal to MDA developers while at the same time providing sufficient precision to design and implement tool support. The test will be whether the industrial partners in the project and the greater MDA community as a whole will find the definition of refinement suitable for their tasks, and usable in large-scale modelling and refinement projects.

References

[1] Object Management Group, "Model Driven Architecture official web-site". URL: http://www.omg.org/mda.

[2] Object Management Group, official web-site. URL: http://www.omg.org/.

[3] Bast, W., A. Kleppe, and J. Warmer, "MDA Explained: The Model Driven Architecture: Practice and Promise", Addison Wesley, 2003.

[4] Fowler, M., "Platform independent malapropism". URL: http://www.martinfowler.com/bliki/PlatformIndependentMalapropism.html

[5] Ambler, S.W., "A road map to Agile MDA". URL: http://www.sdmagazine.com/documents/s=826/sdm0406e/

[6] "Meta Object Facility Queries-Views-Transformations". URL: http://neptune.irit.fr/Biblio/qvt specification.shtml

[7] Object Management Group, "XMI specification". URL: http://www.omg.org/technology/documents/formal/xmi.htm

[8] Fowler, M., "Model Driven Architecture". URL: http://www.martinfowler.com/bliki/ModelDrivenArchitecture.html.

[9] Mellor, S.J., "Agile MDA". URL: http://www.omg.org/mda/mdafiles/AgileMDA.pdf.

[10] Object Management Group, "Meta Object Facility". URL: http://www.omg.org/mof/.

[11] Modelware IST Project, URL: http://www.modelware-ist.org.

[12] Object Management Group, "UML Specification 1.5". URL: http://www.omg.org/uml/.

[13] Hehner, E.C.R., "A Practical Theory of Programming (Second Edition)", Springer-Verlag, 2003.

[14] Morgan, C.C., "Programming from Specifications (Second Edition)", Prentice-Hall, 1994.

[15] D'Souza, D., and A. Wills, "Objects, Components and Frameworks with UML", Addison-Wesley, 1999.