Chapter in “Model-Driven Software Development, Volume II”, S. Beydeda and V. Gruhn editors, Springer Verlag

A Generalized Notion of Platforms for Model Driven Development

Colin Atkinson[1] and Thomas Kühne[2]

[1] University of Mannheim, 68161 Mannheim, Germany colin.atkinson@ieee.org
[2] Darmstadt University of Technology, 64289 Darmstadt, Germany
kuehne@informatik.tu-darmstadt.de

1 Introduction

Model driven development (MDD) is based on the principle of separating the description of an application's abstract properties and logic from a description of its platform specific implementation, and of automating the transformation of the former into the latter using advanced Model Transformation Tools (MTTs). The most mature formulation of this vision at present is the OMG's “Model Driven Architecture” (MDA) which refers to a high-level description of an application as a platform independent model (PIM) and a more concrete implementation-oriented description as a platform specific model (PSM). Figure 1, taken from the OMG's MDA Guide [OMG03] provides a “suggestive” picture of the MDA vision by illustrating how PIM's are automatically transformed into PSMs with the help of additional input describing the properties and services offered by the target “platform”. While this is undoubtedly a very powerful and elegant metaphor for software development, there are some significant issues which need to be sorted before this vision becomes a reality in mainstream software engineering. Chief among them is the question of what precisely a platform is and what a platform model looks like. Of the three fundamental ingredients of MDA referred to in Fig. 1, “model”, “transformation” and “platform”, “platform” is currently the vaguest and least well-defined. This may be because most of the research on MDA to date has focused on the “transformation” challenge [Met05, GGZ+05, SPGB03]. There are few explicit definitions of the notion of “platform” or platform model in the MDA literature and those that do exist are rather vague and high level. However, without a precise and concrete definition of what a platform is and what a platform model looks like it is impossible to formulate a precise notion of PIMs, PSMs and the additional “input” depicted in Fig. 1.

The most explicit definition of the concept of a platform in the MDA context is probably to be found in the MDA Guide [OMG03], which states:

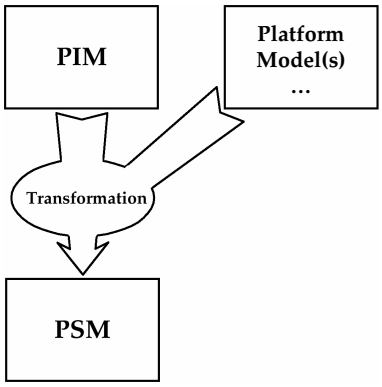

Fig. 1. Core Principle of MDA

> *"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented."*

Although this is a very high-level definition which leaves a lot of scope for interpretation, it does make it clear that a platform is intended to be viewed as a vehicle for the *execution* of a software application. This is reinforced by the "flagship" text that accompanies the MDA logo on the OMG's website[3]:

> *"Platform-independent applications built using MDA and associated standards can be realized on a range of open and proprietary platforms, including CORBA, J2EE, .NET and Web Services or other Web-based platforms."*

This is no accident of course, because the word "platform" is generally used in the IT industry to refer to machines or systems that are built to support the execution of software applications in their end-user environment. In addition to the platforms already mentioned, examples include hardware such as the Intel Pentium or Power PC processors, operating systems such as Linux or Windows, and virtual machines such as the Java Virtual Machine and the .NET Common Language Runtime. If one views a platform as an execution infrastructure, it seems self evident that a "platform model" is a "model of an execution infrastructure". However, this is as concrete a definition as one can extract from the OMG's MDA literature, which is clearly unsatisfactorily vague to drive the transformation from PIMs to PSMs. The most concrete definition of "platform model" available today comes from the school of thought that characterizes MDA in terms of transformation between

[3] http://www.omg.org/mda/

domain specific languages (DSLs) [CKM+99, CK03, GS03]. According to this school of thought, the essential difference between the input and output models in the MDA transformation illustrated in Fig. 1 is that they are written in different languages (or languages dialects). In other words, the information that has to be input into the MTT to effect the transformation is a description of the languages that the models are written in. Therefore, although it is not stated explicitly, language definitions essentially play the role of platform models in the DSL view of MDA. Thus, if one takes a DSL interpretation of Fig. 1, the PIM is written in one DSL, the PIM in another DSL, and the platform model(s) input to the MTT is a description of one or both of these DSLs. In this chapter we challenge both of these implicit assumptions that underpin the generally accepted notions of "platform" and "platform model", namely the "platform = execution infrastructure" assumption and the "platform model = language definition" assumption. In the next section we start by discussing the traditional view of a platform as an execution vehicle and identify the various elements from which a platform is typically constructed. Section 3 follows with a description of the traditional ways in which these platform elements are described. In Sect. 4 we then present a more general notion of "platform" which is sufficient to provide a sound foundation for MDA. Finally, in Sec. 5 we discuss the ramifications of this model for the MDA transformation approach depicted in Fig. 1 and the model transformation tools which are expected to enact them.

2 What is a Platform?

In the context of MDD, as mentioned above, a platform is regarded as "a set of subsystems and technologies" that provide the capabilities needed to support the execution of a software application. Because of their complexity, modern execution platforms are generally visualized and organized as a hierarchy of layers.

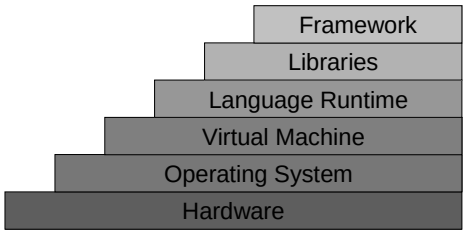

Fig. 2. Typical Platform Layers

Fig. 2 gives a schematic view of the layers that one typically finds in an execution platform. A software application may have access to all the layers

in this hierarchy or some of the layers may completely hide layers below them. For instance, an operating system may make some aspects of the underlying hardware completely transparent (as in the "DirectX" video driver scheme), or a virtual machine may completely abstract away the choice of an operating system (as accomplished in Smalltalk [GR83]). The notion of a platform that is relevant for a particular application depends on the set of layers that it makes use of. For instance, if a virtual machine is not able to completely hide the idiosyncrasies of an operating system's file system or scheduling policy, and these features are utilized by an application, then these aspects of an operation system need to be part of the application's perception of its execution platform. In the following we consider each of these elements in turn.

2.1 Hardware

Computer hardware is the most basic kind of platform on which a computer application can run, and forms the base of the hierarchy of layers. A hardware platform makes a set of basic capabilities available to applications in terms of a set of machine language instructions, memory, and various assisting components, such as a floating point arithmetic unit, etc. Thanks to compiler technology, these rarely if ever have to be considered in application development. However, sometimes the performance criteria of a platform may only be met if certain special-purpose hardware (such as a digital signal processor) is used. In such cases, these hardware capabilities have to be included in the platform model.

2.2 Operating System

The second layer in the hierarchy of platform elements is usually regarded as being the operating system. This provides a whole host of additional capabilities such as file systems, processes, threads, etc. Operating systems are rarely regarded as covering or hiding the underlying hardware because they do not reproduce the execution capabilities offered by the hardware, but rather augment it with many additional services. It is quite common for software applications to depend on the specific capabilities offered by an operating system just as much as on the underlying hardware. For example, one often speaks informally of the "Wintel" platform. Thus an operating system also needs to be part of an application's perception of its platform.

2.3 Virtual Machine

Not all platforms have a virtual machine layer, but if present, this layer is typically regarded as being on top of the operating system layer. The role of a virtual machine is to make the actual choice of operating system and hardware transparent. By doing so it obviates the need to compile all parts

of the application down to the layer of the operating system and hardware. A virtual machine may completely hide the underlying layers from the layers above, or it may let some of the underlying layer's functionality or properties shine through.

2.4 Language Support

Any layer above the virtual machine layer, or in its absence the operating system layer, assumes a certain language which library, framework or application are expressed in. While most of the support for executing the language is provided by the virtual machine or hardware, the compiler adds important functionality in the form of a language runtime system that is a part of the execution platform. Also, high-level language constructs made available to layers above are expressed as templates of low-level byte code or machine code. In addition, predefined values available to programmers may be supported by an underlying virtual machine but are part of, and are typically generated by, the corresponding language support. The classic "runtime" system that compilers bundle with their application hence also needs to be counted as "language support".

2.5 Libraries

Object-oriented programming languages like Java, C++ or Smalltalk typically come with a rich set of libraries providing additional, predefined functionality. Some of these libraries are regarded as standard, and must be present in any platform that aims to support the language, while other are "optional" and provide solutions only for specific domains or purposes. Libraries are the basic mechanism by which middleware technologies that form the foundation of distributed platforms are realized. All of the main capabilities associated with middleware technologies, ranging from "remote message interchange" and "transaction support" to "components and services" are made available to developers in the form of so called "Application Programming Interfaces" (API's) as libraries. Thus, the libraries upon which an application depends form an essential part of its perception of its platform.

2.6 Framework

In contrast to libraries, which can be regarded as passive building blocks for the assembly of software, frameworks contain active control code. They prestructure applications built using them according to some standard control scheme and provide standard solutions for a family of applications. A middleware solution featuring services like "transaction control" or cleverly managed persistence which embodies certain standard usage styles, is much more akin to a framework than a library. The special utility of a framework in providing

an execution infrastructure over and above a library is not only the generic control code but also the design of how to use parts of the framework (or a library) in a generic way across a family of applications. Technically, one could subsume frameworks under "Libraries" as well, since they also represent standard, predefined code. However, if we want to characterize the purpose and nature of the predefined code, it makes sense to distinguish between library and framework layers.

3 Describing Platforms

Having discussed the various elements that comprise a modern execution platform, in this section we now discuss what information needs to be captured to fully characterize a platform from the perspective of an application.

3.1 Hardware

A hardware platform is characterized by the set of instructions, memory model and further functionality that it supports. In general, some form of assembler language plus a description of available hardware components is sufficient to formally capture the properties of a particular type of hardware platform.

3.2 Operating System

An operating system is characterized by the set of services or so-called "system calls" that it makes available to applications. Although these services are ultimately realized in terms of routines implemented in the underlying hardware platform, application programmers, or users interacting directly with the operating system, invariably invoke them using a high-level linguistic representation. Therefore, the definition of the system calls supported by an operating systems usually involves the use of a language to describe the signatures of the calls that can be invoked. For instance, in the case of the Unix operating system this interface (and implementation) language is "C". Furthermore, an operating system generally offers standard instances which applications can use. UNIX , for example, offers instances such as */dev/null*, */dev/zero*, or a system timer.

3.3 Virtual Machine

The description of a virtual machine is usually more complex than that of an operating system because it involves the definition of some of the programming language features as well as the predefined system calls that the language environment provides. In other words, it needs to cover both hardware and operating system layers plus an additional language support layer (see below).

3.4 Language Support

The description of a (programming) language is generally captured in terms of four main elements:

1. Abstract Syntax
2. Concrete Syntax
3. Static Semantics (or well-formedness rules)
4. Dynamic Semantics (or behavior semantics)

Usually the static semantics (3) is checked prior to execution (e.g., by a compiler), so it is not normally necessary to include it in the description of the runtime infrastructure, i.e., the platform model. Likewise, the concrete syntax (2) typically plays no role at runtime, as it will have been compiled to byte code, or machine code prior to execution. In some cases, one might rely on runtime interpretation or compilation of (program generated) source code, in which case 2 & 3 become relevant again for a complete platform description.

3.5 Library and Framework

Apart from "native classes", which are really part of the underling language cast in terms of library elements, the library and framework aspects of a platform are invariably expressed using the features of the language support by the language support and virtual machine layers.

4 Platforms and Platform Models for MDA

Having discussed how platforms are typically structured and described in traditional IT technology, we are now in a position to explain how we believe these approaches should be generalized to support the notions of "platform" and "platform model" in MDD. In doing this we have four goals in mind. We want to provide a notion of platform and platform model that –

- is consistent with OMG MDA terminology
- accommodates the approaches describes in the previous two sections
- is complete
- is composed of orthogonal concepts, avoiding redundancy and overlap

To achieve these goals we need to move away from two of the fundamental assumptions which implicitly underpin the notions of platform and platform models in the MDA literature today, namely the assumptions that

1. platform = execution infrastructure
2. platform model = language definition

Although platforms are most commonly thought of as execution infrastructure in the IT industry, this is not always the case. In general, a platform can be viewed as any system capable of supporting the fulfillment of some goal with respect to a software application. Executability is certainly one important goal, but it is not the only one. Next to execution, useful goals to support are checking, presentation and even the creation of models, i.e., model development. Thus, the concept of a platform in its full generality also embraces what is typically viewed as the development environment as well the execution environment. In this sense, a system which supports the model-based representation of an application as part of development is just as much a platform as systems that support its execution, testing, validation (see [GTS05]) or benchmarking for example. Generalizing the concept of "platform" to encompass both *deployment* and *development* environment brings several benefits as explained below. Ideally, a model of a platform should provide a complete and accurate description of that platform, so that any application that is consistent with the platform model is guaranteed to be processable by that platform. Although language definition is an important element of such a platform definition, as discussed in the previous section it is not by itself enough. There are aspects of a platform that are not captured using the classic language description techniques, such as the functionality provided as system calls and libraries. Thus, the concept of a platform model needs to be extended to include other elements.

4.1 Generalized MDD Platform Model

Analyzing the various platform elements and description techniques discussed in Sects. 2 & 3 we can identify four basic facets through which information about the capabilities and rules of an object-oriented platform is conveyed.

Language The first facet is a language facet which describes the basic concepts with which applications designed to use the platform can be constructed. This of course, corresponds to the language support element of the platform description techniques discussed in Sec. 3.

Predefined Types The second facet consists of a set of predefined types (e.g., classes) which augment the core language capabilities with additional services. This corresponds roughly to the library element in traditional platforms, but includes predefined types coming from the language support layer (e.g., class "Object" in Java).

Predefined Instances The third facet consist of a set of predefined instances (e.g., objects), which are ready to be used out of the box. This facet contains the pre-instantiated objects that are found in some libraries such as Java's standard I/O streams "in, out, err" or Smalltalk's "true" and "false" instances, but also preexisting system timers, etc. Furthermore, if predefined operations are thought of as belonging to a single unified system (in the sense of systems calls) this is the facet which contains the system as a pre-instantiated object ready to receive system calls.

Patterns This fourth facet consists of the additional concepts and rules that are needed to use the capabilities found in the previous facets in a meaningful fashion. We use the terms "patterns", but in general any "usage rules", such as those that characterize a framework, are found in this facet. Typically, software may be used in many contexts and ways and displays different properties accordingly. In order to achieve exactly the functionality and properties intended for a particular standard platform, one must associate the corresponding usage patterns with the platform. A complete platform description therefore not only needs to specify the available parts but also the intended ways in which to use these parts. This knowledge can be expressed in a variety of ways, including as patterns in the style of design patterns [GHJV94].

In Fig. 3 we represent all the facets including their role by representing patterns as an oval covering the three other facets. The language facet sits above "Types" and "Instances", since it is the defining layer for both. Layer "Language" corresponds to language-*definition*, whereas "Types" and "Instances" correspond to language-*usage*. We refer to this view of a platform as the General Platform Model (GPM). We only show two logical metalevels (types and instances) in this picture since most mainstream languages do not offer more then two levels of language use. In general, Fig. 3 may feature further facets below "Language", such as "Metatypes", etc.

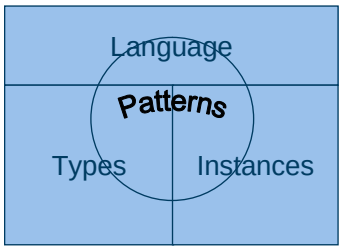

Fig. 3. General Platform Model

An important point about the general platform model illustrated in Fig. 3 is that it is not intended to correspond to a cumulative collection of platform model elements as discussed in Sect. 2, but rather is intended to provide a way of characterizing *each* element. Each element may place a different emphasis on the different platform facets, but can nevertheless be expressed using the same overall notion of platform. For example, an operating system, which offers most of its capabilities in the form of system calls, will have a platform model that is predominantly centered on the instance facet. On the other hand, a virtual machine, which offers a large proportion of its capabilities in the form of a language, will naturally have a platform model that is predominantly centered on the language facet. It is our contention that

any kind of platform can be modeled through the appropriate combination of these facets, and in fact, is typically incomplete if one more of the facets are ignored. In Fig. 4 we illustrate this by showing how each of the traditional platform notions discussed in Sect. 2 can be represented in terms of the GPM.

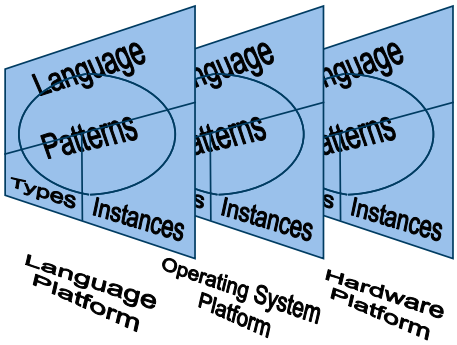

Fig. 4. Full Platform Description

4.2 Stack Example

To illustrate that these four facets are indeed sufficient and minimal let us consider how a small example would be modeled using the GPM. Figure 5 shows a highly simple Stack class written in Java.

According to our approach, this Stack is written based on a model of the platform that is going to execute it – namely the Java Virtual Machine and its predefined execution environment – after compilation by the standard JDK compiler. To ascertain what aspects of the platform are important for this application we simply have to ask whether a potential change to the way in which the application is represented would render it non-executable by the target platform. Clearly the features of the core Java language are important (language facet), because if we were to use any non-Java features the program would no longer compile. In the extreme case we could write the code in another language like C++, but then the application would be targeted to a different platform and would not be executable on the Java platform. Another important feature of the stack class is its use of the utility class "Vector" from the predefined Java library (types facet). This is clearly another dependency on the Java platform, since if this class were not available, or were given a different semantics to that expected, the application would not run or would not run as expected. This dependency has nothing to do with the basic language (language facet), however. It would be perfectly possible to define another platform based on the same Java core language but with a different library of predefined classes. Another dependency of the stack class

```
import java.util.Vector;

public class Stack {
     protected int max, elems;
     protected Vector entries;

     public Stack (int maxValue) {
         max = 10;  elems = 0;
         entries = new Object[max];
     }

     public void push (Object o) throws Exception {
         if (elems < max) {
             entries.addElement(o);
             elems++;
         } else {
             System.out.print ("Stack is full")
             throw new Exception("FullStackException");
         }
     }

     public Object pop () throws Exception {
         if (elems >= 0) {
             return entries.lastElement();
         } else {
             System.out.print ("Stack is empty")
             throw new Exception("EmptyStackException");
         }
     }
}
```

Fig. 5. Java Stack Example

on the Java runtime environment is its use of the standard output stream "out" to output messages (instances facet). Although the stream is accessed via the library class System, in effect, "out" references a predefined stream object. As with the predefined classes such as Vector, the absence of the standard output stream, or a change to its semantics, would change the Stack's ability to execute on the platform, or would change its intended effect.

Fig. 6 illustrates how the various elements of the standard Java platform can be represented according to the GPM approach. For simplicity we have refrained from extending the example to feature a full application of a "Handle Body" pattern, such as the Bridge pattern [GHJV94], and therefore left out the patterns facet. As illustrated by this example, a platform changes if any one of the facet elements in the GPM changes. Thus, a change to the types or to the instances results in a different platform even if the core language remains the same. In fact, this is precisely how Java, as a general technology

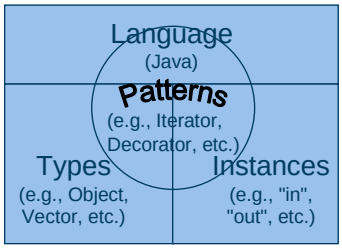

Fig. 6. GPM Representation of the Java Platform

foundation, has been adapted by Sun to support the many middleware and enterprise technologies that it is now known for. The core Java language remains untouched whenever possible, but the set of predefined set of types and instances is extended or changed. Figure 7 shows how the J2EE platform is defined by adding additional predefined types and instance features to those available in the basic J2SE platform, leaving Java as the base language in the language facet. In addition, the platform contains a well-defined set of new patterns which describe how these types and instances should be used.

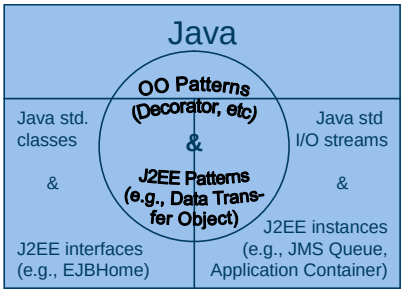

Fig. 7. J2EE Platform

An advantage of explicitly separating the distinct aspects of a platform in a GPM is that the relationship between high-level and intermediate-level representations of applications is clarified. Java technology, for example, actually defines two platforms: The high-level language platform in which application code is represented in the Java high-level language (like the Stack class in Fig. 5), and the byte code platform in which application code is represented in Java byte code. Thus, a full model of standard Java (J2SE) technology would include two GPMs: One describing the capabilities used by application developers based on the Java high-level programming language and the other describing the capabilities of a Java virtual machine in terms of Java byte code. A Java compiler can then be understood as a very specialized model transformation tool which maps models written according to one GPM into

models written according to another. As a more complex example we consider how the GPM might be used to model the various language and capabilities in the ".NET" Platform. The generic term .NET encapsulates a wide range of development technologies ranging from enterprise servers to web services. However, the core of the technology is the so called .NET framework which provides a variety of different execution platforms. The basic architecture of this family of platforms is described in an ECMA standard, known as the Common Language Infrastructure (CLI) [CLI02]. One of the key goals of this standard is to make it possible for code written in different languages to interoperate. To achieve this, the CLI defines a language-spanning type system, known as the Common Type System (CTS) and a core set of features (the Common Language Specification (CLS)) that all compliant languages must support. Code written in a language that is compliant with the CLS is referred to as managed code and is guaranteed to be able to execute all .NET features and to interoperate with all other managed code, regardless of the managed language used to write it. Figure 8 illustrates how the GPM can be used to represent the family of platforms encapsulated by the .NET framework.

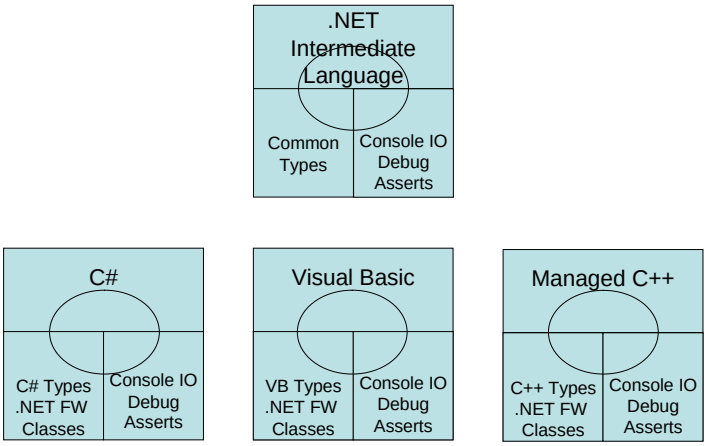

Fig. 8. .NET Platform Family

The GPM at the top of Fig. 8 represents the low-level execution platform defined by the Common Language Infrastructure in terms of the Common Type System and the intermediate language. This is implemented as the Common Language Runtime. The lower three GPM's in Fig. 8 represent some of the different managed language environments currently implemented. The only aspect that differs in these GPMs is the base language in the language facet and the availability of extra language-specific types, such as Visual Basic specific types in the second GPM, C++ specific types in the third GPM, etc. Several of the GPMs in addition contain specific features in their instance

facet. For example, the C++ GPM contains the system instances as an abstraction of the underlying runtime system responsible for realizing system calls, the J# GPM contains the usual standard I/O instance objects such as "in" and "out", etc. Furthermore, each of the platforms in the platform family defines a set of patterns (usage guidelines) for the use of the common and specialized features.

Most of the ".NET" features are wrapped up in the predefined type library captured in the type facet of the GPM. In the case of .NET, the framework library contains a vast collection of classes which provide a wide range of capabilities ranging from basic network and middleware services through to features for database access, dynamic web page generation, GUI development and web service creation.

5 Model Transformation Tools

In the previous section we introduced the concept of the General Platform Model, and explained how it can be used to model all of the different kinds of execution platforms encountered in MDD. We also introduced the notion that all environments which perform some useful function with respect to a software application should be regarded as platforms and characterized using GPMs, not just execution platforms. In this section we discuss the ramifications of these ideas on the MDD vision, and explain how they help resolve the issues identified in the introduction.

The MDA literature usually presents the core transformation concept as if platform specificity or independence of the input and output models were a binary property. For example, in Fig. 1, the input model is referred to as "platform independent" and the output model as "platform specific", but there are two reasons why it is inappropriate to think of platform independence/specificity in this black or white manner given the current "platform = execution infrastructure" assumption of MDA:

First, it only makes sense to speak of platform independence and platform specificity as binary properties in such a situation if one has an "ideal" MTT which can perform the transformation from complete platform independence to complete platform specificity in *one* step. Otherwise, the many steps required to arrive at the bottommost platform specific model, starting from a high-level platform independent model, automatically introduce shades of platform specificity. However, such an ideal tool is a long way from realization.

Second, even if such an ideal MTT were available it might still be desirable to produce the models at intermediate levels of abstraction which highlight a particular aspect of the architecture or reveal a certain aspect of the application [Ham05, Wai05, AK03]. Thus, for the foreseeable future, the MDA transformation step illustrated in Fig. 1 is likely to be applied in the context of a chain of transformation steps, each creating a model of the application which is closer to the final execution platform than the previous model. Only

the first and last application models in such a chain would then be characterizable as either (fully) platform independent or (fully) platform specific, and the intermediate models would have a certain degree of platform independence/specificity which lies somewhere between. This view is in fact explicitly acknowledged in the MDA Guide which states that:

> *"Platform independence is a quality, which a model may exhibit. This is the quality that the model is independent of the features of a platform of any particular type. Like most qualities, platform independence is a matter of degree."*

Given this situation, therefore, Fig. 9 represents the currently prevailing view of how MDA transformation technology will be used in practical software engineering scenarios.

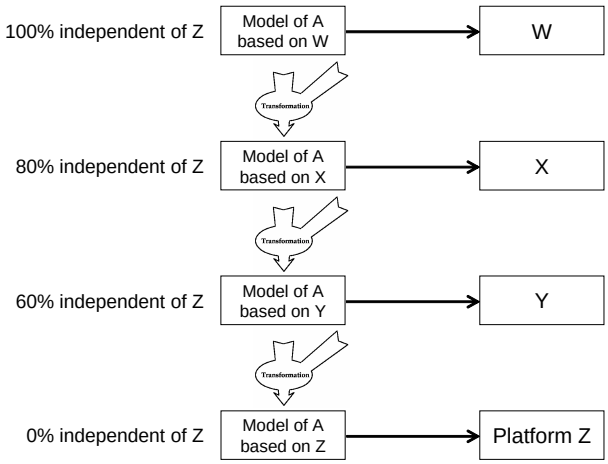

Fig. 9. Existing View of the MDA Transformation Chain

Several aspects of Fig. 9 are worthy of note. First, although all of the application models in the chain are "based on" (i.e. represented in terms of) something, only the bottom model is based on an actual executable platform. According to the "platform = execution infrastructure" assumption, X, Y and Z upon which the other models are based are not platforms, since the associated models are not yet executable. The question is: If they are not platforms, what are they? The DSL school of MDA would answer that these are DSLs. However, as explained in the previous sections, DSLs described using the traditional language definition techniques are not in general sufficient to describe all the characteristics/facets which such models might need. In particular, they do not capture type libraries, instances and patterns.

Fig. 10 shows the alternative view of the model transformation chain based on the proposals put forward in this chapter. The main point to note is that

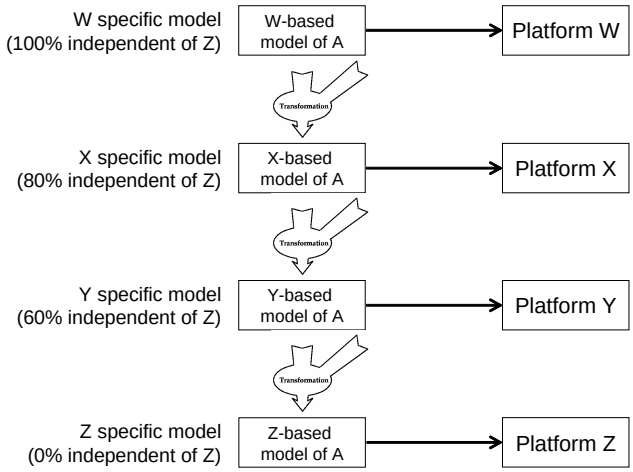

Fig. 10. Proposed View of the MDA Transformation Chain

W, X and Y on the right hand side of the picture are now also viewed as platforms, not just Z. They may not be platforms supporting execution but in our approach they would nevertheless be considered platforms and would ideally be represented as GPMs. This in turns means that all of the application models can be viewed in one sense as platform specific models because they are all based on (and thus 100% specific to) the platform they are written in terms of. It is still of course possible to assign each model a measure of independence or specificity with respect to another particular platform, such as the ultimate target platform model for the which the application is being developed. But by requiring every model to be "based on" exactly one platform model, the original terminology and intention of the fundamental MDA transformation step once again becomes meaningful. Every model is specific to one platform and (relatively) independent of all the others. The final question which needs to be addressed is how the proposed model shapes the additional input that drives the transformation. Because the MDA literature uses the phrases "independence" and "specificity" in relation to the *output* model of the basic transformation step, there is an implication that MTT's will be specific to the *input* model platform but parameterized with respect to the *output* model platform. Such an MTT might, for example, be tied to the UML platform as the base for its input models but parameterized with respect to the platform model of its output. This situation is illustrated in Fig. 11, which enhances Fig. 10 with information about the input models driving the transformation step.

In Fig. 10, it is the model of the platform upon which the output model is based, that serves as the additional input to the transformation. This represents only one point on a spectrum of possible tool parameterization, however. Other tools which provide a different balance are also feasible. At one

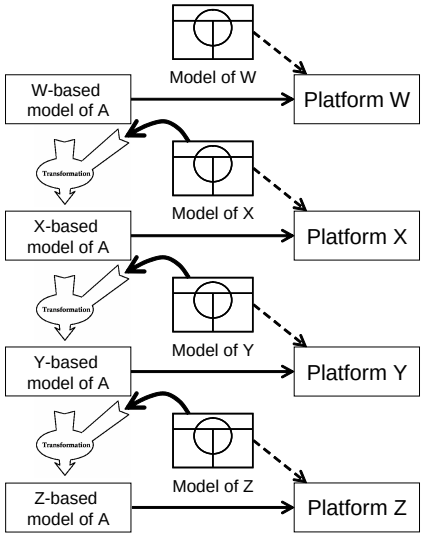

Fig. 11. Parameterization by Ouptut Platform Models

extreme, there are MTTs which are specific to both the input and the output platforms, and cannot handle models which are targeted to other platforms. Today's compilers are examples of this kind of MTT. At the other extreme, one can envisage MTT's which are fully parameterized with respect to both the input and output platform models as illustrated in Fig. 12.

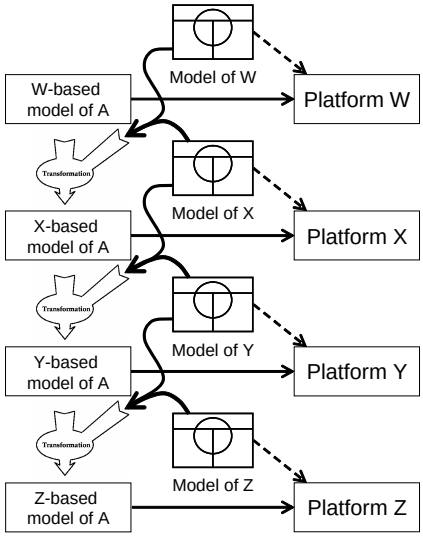

Fig. 12. Parameterization by Input & Output Platform Models

For the foreseeable future it is likely that we will see MTT's which occupy the full spectrum of genericity between these two extremes, as illustrated in Fig. 13. Initially one can expect to see MTT's that are similar to compilers and are hardwired to transform between two specific platform models, but gradually more generic tools will be developed which will be parameterized with respect to increasingly more aspects of the target *and* source platforms.

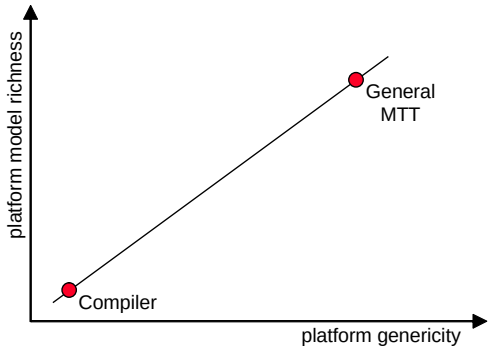

Fig. 13. Spectrum of Transformation Tools

6 Summary

In this chapter we have identified two significant problems with the notion of platform and platform model in the current vision of MDA technology and have suggested a possible approach for solving them. The first problem is that the concept of platform promoted in the MDA literature is strongly associated with the notion of execution infrastructure, and thus implicitly rules out the consideration of other kinds of environments as platforms. The second problem is that the prevailing vision of MDA as a vehicle for supporting families of domain specific languages (DSLs) leads to a de facto way of representing platforms (or descriptions that play the roles of platforms) that is not rich enough to capture all the characteristics of platforms as they have traditionally been viewed in the IT industry. Our proposed solution is to expand the set of environments that are considered platforms to include those that offer development time capabilities, such as model validation, and to generalize the notion of a platform model to include all information that is needed to capture the necessary features of platforms. Such General Platform Models (GPMs) include information about the predefined types, predefined instances and usage patterns that characterize a platform *in addition to* information about the language features supported, as is currently the case.

7 Acknowledgements

The authors would like to thank Clemens Szyperski for clarifying discussions on the ".NET" platform family.

References

[AK03] Colin Atkinson and Thomas Kühne. Aspect-oriented development with stratified frameworks. *IEEE Software*, 20(1):81–89, 2003.

[CK03] Steve Cook and Stuart Kent. The tool factory. OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture, 2003.

[CKM+99] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. Defining UML family members using prefaces. In Christine Mingins, editor, *Proceedings of TOOLS Pacific 1999*. IEEE Computer Society, 1999.

[CLI02] *Standard ECMA-335, Common Language Infrastructure (CLI), 2^{nd} edition*, 2002. ISO/IEC 23271, http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[GGZ+05] Lars Grunske, Leif Geiger, Albert Zündorf, Niels Van Eetvelde, Pieter Van Gorp, and Daniel Varro. Using graph transformation for practical model driven software engineering. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[GHJV94] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Object-Oriented Software Architecture*. Addison-Wesley, 1994.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[GS03] Jack Greenfield and Keith Short. Software factories: Assembling applications with patterns, models, frameworks and tools. OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture, October 2003.

[GTS05] Aniruddha Gokhale, Gabriele A. Trombetti, and Douglas C. Schmidt. A model-driven development environment for composing and validating distributed real-time and embedded systems: A case study. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Ham05] Imed Hammouda. A tool infrastructure for model-driven development using aspectual patterns. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[Met05] Andreas Metzger. A systematic look at model transformations. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

[OMG03] OMG. *MDA Guide Version 1.0.1*, 2003. Version 1.0.1, OMG document omg/03-06-01.

[SPGB03] S. Sendall, G. Perrouin, N. Guelfi, and O. Biberstein. Supporting model-to-model transformations: The VMT approach. Workshop on Model Driven Architecture: Foundations and Applications; Proceedings published in Technical Report TR-CTIT-03-27, University of Twente, 2003.

[Wai05] Gabriel A. Wainer. A model-driven technique for development of embedded systems. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-driven Software Development – Volume II of Research and Practice in Software Engineering*. Springer, 2005.

Index