Model Driven Engineering

Stuart Kent

University of Kent, Canterbury, UK
sjhk@ukc.ac.uk
http://www.cs.ukc.ac.uk

Abstract. The Object Management Group's (OMG) Model Driven Architecture (MDA) strategy envisages a world where models play a more direct role in software production, being amenable to manipulation and transformation by machine. Model Driven *Engineering* (MDE) is wider in scope than MDA. MDE combines *process* and *analysis* with architecture. This article sets out a framework for model driven engineering, which can be used as a point of reference for activity in this area. It proposes an organisation of the modelling 'space' and how to locate models in that space. It discusses different kinds of mappings between models. It explains why process and architecture are tightly connected. It discusses the importance and nature of tools. It identifies the need for defining families of languages and transformations, and for developing techniques for generating/configuring tools from such definitions. It concludes with a call to align metamodelling with formal language engineering techniques.

1 Introduction

This article describes a conceptual framework for the model driven engineering (MDE) of software intensive systems. It begins, in section 2, with an overview of the OMG's Model Driven Architecture (MDA), and identifies a number of aspects of the model driven approach which MDA has yet to address. The remainder of the article discusses those aspects. Section 3 identifies various dimensions of the modelling space other than the platform independent (PI)/platform specific (PS) distinction made by MDA. Section 4 discusses mappings between models, including mappings from PI models (PIMs) to PS models (PSMs). Section 5 explains why process can not be divorced from architecture, and why, therefore, definition of process is just as central to the model driven approach. Section 6 discusses the importance of tools to MDE, and considers various kinds of tooling. Section 7 argues that MDE requires meta techniques and technology that allow us to define families of languages and of processes, and tooling to boot. Section 8 discusses what technologies exist and what they may become.

2 Model Driven Architecture

A detailed description of the OMG's MDA strategy is provided in [16]. This begins as follows:

> "The OMG's mission is to help computer users solve integration problems by supplying open, vendor-neutral interoperability specifications. The Model Driven Architecture (MDA) is OMG's next step in solving integration problems."

On page 2, [16] further elaborates on the problem:

> "There are limits to the interoperability that can be achieved by creating a single set of standard programming interfaces. Computer systems have lives measured in decades, and not all ancient systems written in obsolete programming languages can be modified to support standards. Furthermore, the increasing need to incorporate Web-based front ends and link to business partners who may be using propietary interface sets can force integrators back to the low-productivity activities of writing glue code to hold multiple components together. When these systems in their turn need modifying and integrating with next year's hot new technology (and they will) the result is the kind of maintenance nightmare all computer users fear."

Then on page 3, section 2.1, it indicates how MDA aims to solve the problem:

> "The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. [] The MDA approach and the standards that support it allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms, and allows different applications to be integrated by explicitly relating their models, enabling integration and interoperability and supporting system evolution as platform technologies come and go."

This identifies the key distinction made in MDA, between platform independent and platform specific models (PIMs and PSMs). Much of the rest of [16] is an attempt to explain this distinction. The claim is that by abstracting away from platform-specific detail (pages 7 & 8) "it is easier to validate the correctness of the model", "it is easier to produce implementations on different platforms while conforming to the same essential and precise structure and behaviour of the system", and "integration and interoperability across systems can be defined more clearly in platform-independent terms, then mapped down to platform specific mechanisms".

So what are PIMs and PSMs? The definition is given in [16] on page 6: "A PIM is a formal specification[1] of the structure and function of a system that abstracts away technical detail." A PSM is not a PIM, but is also not an implementation. It is a "specification model of the target platform". These

[1] [16] page 3: "A specification is said to 'formal' when it is based on a language that has a well-defined form ("syntax"), meaning ("semantics"), and possibly rules of analaysis, inference, or proof for its constructs. [] The semantics might be defined, more or less formally, ..."

specification models may have different bindings to particular implementation language environments (like CORBA). The functionality specified in a PIM is realized in a platform-specific way in the PSM, which is derived from the PIM via some transformation.

On page 13, [16] expands on different kinds of mappings that might be involved in MDA: PIM to PIM (refinement of designs), PIM to PSM, which has already been discussed, PSM to PSM (refinement of platform models) and PSM to PIM (mining of existing implementations for useful PI abstractions). However. most of the paper focuses on the PIM to PSM mapping.

It is worth noting that there may be many PIMs which may take different viewpoints or perspectives on the system and which may be more or less abstract. This is acknowledged by [16]. However MDA itself, at least as defined in [16], is reasonably quiet about how that space might be structured, other than distinguishing between PIMs and PSMs. One goal of this article is to provide a more general characterisation of the modelling space.

A clear goal of MDA is to provide a framework which integrates the existing OMG standards. Thus it comes as no surprise that it is intended that all PIMs and PSMs be expressed in the Unified Modeling Language (UML), using its profile mechanism to specialise and extend the language for different contexts (it is unlikely that exactly the same language will fulfill all the needs of every kind of PIM and PSM). The Meta Object Facility (MOF) will be used to define the UML and its profiles not only in a formal way, but also in a way that supports the generation/configuration of tools directly from the definitions. The OMG has recently issued requests for proposals for the revision of MOF so (a) that it is a subset of UML itself (true metamodelling) and (b) that it has facilities for defining transformations between metamodels. We will revisit MOF in the final section of this article, where we discuss the kinds of meta technologies required to realise Model Driven Engineering.

Although MDA does not insist that the mappings between PIMs and PSMs be automated, it is clear that this is the intention. [16] page 13 talks about how mappings might be executed using, for example, scripts in CASE tools or external tools such as XSLT on XMI files[2]; and a more detailed discussion of code generation possibilities is provided on page 15. Pages 24 and 25 discuss the place of existing OMG standards in the context of MDA. XMI and MOF are most relevant to tooling. XMI "adds *Modeling* and *Architecture* to the world of XML", and XML, of course, delivers models in a machine-processable form. Note that XMI is focussed on a representation of the abstract syntax of models, not just surface representation. MOF "defines programmatic interfaces for manipulating models and their instances []. These are defined in IDL and are being extended to Java."

To summarise, MDA focuses on architecture, on artefacts, on models. It aims to exploit the usefulness of models as tools for abstraction, for summarising, and for providing alternative perspectives. Although it acknowledges there might be

[2] XMI is the OMG standard which stipulates how UML and MOF models are rendered in XML.

a richer modelling space, it chooses to focus on just one dimension, the dichotomy between platform independent and platform specific models. It claims that UML is up to the task of defining all PIMs and PSMs, though acknowledges a need for UML profiles to tailor UML to particular modelling contexts. MOF will be used to define languages (chiefly UML profiles) and transformations between languages. A clear goal is that transformations between models should at least be partially automated, thereby reducing the burden of keeping models in step to the point that their intrinsic benefits outweigh the costs of their maintenance.

The remainder of this article explores all those aspects of a model driven approach largely ignored by MDA: dimensions of the modelling space other than the PIM/PSM distinction; transformations or mappings between models including mappings from PIMs to PSMs; the close relationship between process and architecture; various kinds of tools. The article concludes with a discussion of language engineering technologies required to support this vision, which will include some assessment of MOF and UML.

3 Modelling Dimensions

In general, when one produces a model, one needs to take account of the perspective that model is intended to take. One way to structure the modelling space is to categorise perspectives. MDA does this, but it just has two categories: PIMs and PSMs. A more general way to categorise perspectives is to identify the orthogonal criteria which can be used to distinguish between one perspective and another. These criteria can be thought of as different dimensions in an n-dimensional space [22], and a perspective is then at the intersection of different positions along those axes. So what are these dimensions?

Clearly there is the platform specific/platform independent dimension. One could view this as an example of the more general abstract/concrete dimension. In which case we can only talk about whether a model is abstract (platform independent) with respect to some other model. One could imagine a UML specification of an e-business system that abstracted away from details of a specific platform (e.g. J2EE), but which was viewed as a concrete realisation of part of a more abstract model of a business or business process. Perhaps one key feature of the PIM/PSM distinction is that the models are likely to be expressed in different languages[3]. This is not necessarily the case for models which bear an abstract/concrete relationship. For example, refinement is typically a relationship between models in the same language, where the difference between the models is one of granularity of action.

Various dimensions of concern have already begun to be identified in the area of Aspect Oriented Software Development (AOSD) [1]. One dimension of concern is *subject area*, for example the area of the system dealing with customers, or that concerned with processing of orders. Separation of concern by subject area has been proposed in Subject-Oriented Programming [11] and Catalysis [9]. Another dimension of concern has been called *aspect*, for example concurrency control

[3] Here we count different UML profiles as different languages

and distribution. A notable feature of these two dimensions is that often it does not make sense to define mappings directly between models at different locations along a dimension, but rather their relationship is better expressed by showing how they map into one single integrated model. Interestingly, the AOSD community, though accepting that these dimensions may be applied at different levels of abstraction, do not seem to regard the abstract/concrete dimension as just another dimension of the whole space.

A third category of dimension is less concerned with the technical aspects of a system, and more with the managerial and societal aspects. Such dimensions include *authorship*, *version* (as in version and configuration control), *location*, in case the system development is distributed across sites, and *stakeholder*, such as business expert or programmer.

[22] points out that for any particular software development project, one will need to state the dimensions that need to be considered, which includes defining the points of interest along those dimensions. Some of those dimensions are likely to be standard across projects, for example authorship, version and location, only needing to decide whether the dimension needs to be included or not. The subject area dimension is likely always to be present, and that will require the subject areas to be identified. Similarly one will need to decide the relevant points on the aspect dimension, for example whether points are required for concurrency or distribution, for information and data, and so on. One will also need to define the stakeholders involved; and the levels of abstraction are of interest. In MDA, one may choose that the only two points of interest are PIM and PSM along this dimension.

Then, when building a model, one identifies at what intersection of the dimensions the model should be placed. In theory, there could be a different language for describing models at each different intersection. In practice, the language used at a particular point of intersection will be determined by one (or a small subset) of the dimensions that determine that point. For example, a subject area may be defined from the perspective of many different aspects, and the language used will be determined by the aspect being described. It may also be that the level of abstraction and/or stakeholder will influence the language to be used. With regard to the latter, a business expert and programmer are likely to need to see the specification of a system in different languages, even though the two renderings would be isomorphic – they are at the same level of abstraction, looking at the same aspect and subject area.

4 Mappings

We have seen that models can take on different perspectives, and that different perspectives often, but not always, require modelling languages with different properties. Thus, when engineering a particular system, it is possible, in theory, that there may be many models of that system taking different perspectives, and that these models will not necessarily be expressed in the same language (though some might be).

Perspectives on their own are more use than if they can be related, and this requires mappings to be defined. In MDA, the main mapping is between PIM and PSM, and this is often associated with code generation. However, this is not the only kind of mapping required.

There is a distinction between *model translation* and *language translation*. One can define mappings between models in the same language (model translation) and mappings between models in different languages (language translation). Model translations should be expressable in the language (or an extension of the language) in which the models are expressed. Language translation needs to be expressed in terms of the definitions of the languages themselves. Interestingly, if the definitions of the languages are also thought of as models (in terms of OMG standards they would be MOF models, or metamodels) then it should be possible to express language translations in an extension of the language used to define languages, the metamodeling language, which, in OMG MDA, is MOF. Indeed, a request for proposals is soon to be issued by the OMG for just such an extension to MOF. In short, language translation is just (meta-)model translation.

In a model driven approach, it is not enough to leave these mappings implicit, as vague sets of rules. To make it worth the effort of maintaining models other than the target implementation, the burden of keeping models in step needs to be considerably reduced than in current (manual) practice. If models (e.g. target code) can be generated from other models, so much the better. The question, then, is how to define model and language translations so they can be used for the practical purpose of maintaining consistency between models and, where feasible and desirable, generating one model from another.

It is not the aim of this article to come up with a solution to this problem, but it is worth considering what factors need to be take into account. Is the definition intended to be machine processed? If so what kind of processing? Is it to be used to generate one model from another, or to maintain consistency between models? Or should the definition be in a form suitable for mathematical analysis (for example, whether or not it is isomorphic). Another factor to consider is whether or not it is easy to comprehend and/or write, and this depends, of course, on the audience and/or author. For example, one might consider using XSLT, which is machine processable, but might be regarded as difficult to comprehend especially since the models will have to be converted first to XML.

Another option is to try and use the same language (or an extension thereof) as that used to define the models being mapped, in model translation, or the languages being mapped, in language translation. The advantage of this approach is that it is not necessary for the modeller/language designer to learn a new language for writing out mappings. This will be discussed further in section 8.

5 Process

Models and transformations are artefacts. For any given project, a particular set of models providing chosen perspectives on the system will be constructed, and appropriate mappings between these models will be defined. This determines the essential architecture of the artefacts to be constructed during the project. However, this only constitutes part of what is required. Specifically, there needs to be some process, at the very least some guidelines on the order in which models should be produced, how the work should be divided between teams and so on.

There are macro and micro processes associated with the engineering of the different models. Macro processes concern the order in which models are produced and how they are coordinated. Micro processes amount to guidelines for producing a particular model.

A simple example will suffice to illustrate the close relationship between process and architecture. A common approach to software development is to use *use cases.* A use case explores scenarios associated with a particular part, indeed subject area, of the system. Identifying different use cases corresponds to identifying different subject areas. It is possible to define a use case in increments, with the first increment making many simplifying assumptions, and each subsequent increment relaxing assumptions [13]. Defining use cases in this way can help to organise the management of the project. If one adopts an iterative approach to software development, then each iteration can correspond to the completion of a use case or use case increment. The ordering of iterations can be determined by ranking the relative importance of use cases, and by ordering of increments on use cases. In this way, the number of iterations and their ordering is determined by the use cases. Furthermore, the tasks associated with each iteration will be determined to some extent by the choice of models that it has been determined need to be produced in each iteration. It will still be necessary, in addition, to define the micro processes associated with the manipulation of each kind of model in an iteration, and macro processes governing how the development of individual models are to be coordinated.

Thus in general, the definition of the artefacts or models developed by a particular process are intrinsic to the definition of that process. The reverse is also true – it is difficult to see how one can identify the artefacts one needs for a particular project or family of projects, without defining the process within which they are intended to be used. Further, the content of the models (e.g. use cases) can directly influence how the process progresses when it is invoked.

6 Tools

In mainstream practice, models, apart from the code, tend only to get sketched out, sometimes after the fact for documentation, at varying levels of abstraction, and with largely unwritten rules used to (loosely) relate the models to themselves and to the code. This is certainly true for a typical OO development.

In a model driven approach, the vision is that models become artefacts to be maintained along with the code. This will only happen if the benefit obtained from producing the models is considerably more and the effort required to keep them in line with the code is considerably less than current practice. Models are valuable as tools for abstraction, for summarising, and for providing alternative perspectives. The value is greatly enhanced if models become tangible artefacts that can be simulated, transformed, checked etc., and if the burden of keeping them in step with each other and the delivered system is considerably reduced.

Tooling is essential to maximise the benefits of having models, and to minimise the effort required to maintain them. Specifically, more sophisticated tools are required than those in common use today, which are in many cases just model editors. What might these tools be? Here are some examples.

Tools to check/enforce well formedness constraints on models. That is, type/static semantics checking. It still amazes me that many commercially available modelling tools do not do this in any systematic way. This may be because the definitions of languages, such as UML, only pay lip service to these rules.

Support for working with *instances of models.* These include checking the validity of instances against a model, generation of instances from a model, both with and without user interference, and (partial) generation of models from instances. An example is the USE tool [20] which provides some support for working with snapshots and filmstrips [9] which may be viewed as instances of a restricted subset of UML. The tool allows one to create snapshots and filmstrips by hand, then check their validity against a class diagram, annotated with invariants and pre-post conditions in OCL.

These tools are important not least to explore the validity of a model. They provide a means of establishing examples and counter examples for a model, and it is much easier both to elicit and explain a model (e.g. to a customer) in terms of examples. Of course the language(s) used to render examples must be appropriate for the stakeholders for which it is intended.

Tools to support mappings between models. There are a family of different tools here. There are, of course, tools that generate one model from another, the most cited example probably being code generation.

However, there is also a need for *coordination tools*, which would be responsible for coordinating the mappings between different models, by flagging inconsistencies and trying to correct them automatically. For example, such a tool could be used to monitor a refinement relationship (which must be established by the developer as it involves taking design decisions) and checking that the relationship is always a true refinement, or at least maintaining a list of verification conditions that need to be discharged. Another example, would be the translation between PIMs and PSMs. In today's world, many PSMs integrate existing (legacy) code with new code, and this usually requires some parts of the system to be hand coded. Therefore the PSM is being interfered with directly and can not be completely generated. It is therefore necessary to maintain both PIM and PSM, and coordinate their relationship.

A third example is where models are not at different levels of abstraction but still provide alternative perspectives. In that case, each model is likely only to define a part of the system, and needs to be kept in step with the other viewpoints, possibly via some underlying integrated model. We need look no further than this proceedings to see a body of work focusing on this topic. Much of the work on *viewpoints* [15] is also relevant here.

Tools to support *model driven testing*. Here platform specific tests and test data are derived from platform independent models and instances of those models. This could be particularly relevant in the development of today's e-business systems, which often involves the integration of existing systems with themselves and new systems, usually using some kind of messaging infrastructure such as the Java Messaging Service (JMS) or IBM's Message Queuing platforms. In this environment, it would at least be desirable to establish a PIM which provides a definition of the information structures being passed around, and the (invariant) constraints they can obey, which abstracts away from the detail of the component-specific renderings of that information. And then one will need to check that the system does not violate the abstract information model, in particular the invariant constraints. This could be done by monitoring messages that go in and out of ports, then mapping these up to instances of the abstract model, using an appropriate retrieval function, and then checking that the instances satisfy the abstract model. In this case a tool is required to generate abstract instances from concrete instances, which are obtained through probes on the running system.

***Dashboard* applications.** The state of the system is still monitored through probes and mapped to abstract instances, but this time the abstract model is aimed at business experts, and abstract instances are presented in a form that allows those stakeholders to monitor the ongoing business processes which the system implements.

Tools for version control and distributed working. The challenge here is to integrate these tools with multiple modelling and coordination tools.

Tools for managing the software process. For example directing developers in the artefacts they should produce and when, making assessments of the level of quality assurance being adhered to (possibly using data, such as consistency checking results, from other tools) and so on.

An important aspect of realising such tools, is to ensure that they remain flexible and configurable. Observation of practice [14] seems to suggest that developers prefer small, flexible tools that can easily be configured to interwork with other tools. A natural architecture for this tooling would be to follow the architecture of models and mappings between models being used in the project. Frameworks are now beginning to emerge (e.g. Eclipse [17]), which could make this possible. Further, more and more UML modelling tools export models in XML format (actually XMI), which at least allows models to be interchanged with other tools.

7 Roll Your Own: Families of Languages and Processes

A model driven engineering approach must specify the modelling languages, models, translations between models and languages, and the process used to coordinate the construction and evolution of the models. To ensure that the burden of maintaining more than one formal model does not outweigh the considerable benefits of models as tools for abstraction, summarising and providing alternative perspectives, powerful tool support is required.

On the other hand, it is unlikely that the same set of models, mappings and process will fit all domains, all organisations or all projects. There will be variations. This is already evident in standardisation work going on in the OMG. Numerous so-called UML Profiles have been or are being defined for particular domains. See [2] for a complete list. Many may have you believe that these are just slight variations on UML, but don't be fooled. For example, in both the Enterprise Application Integration (EAI) and Enterprise Distributed Object Computing (EDOC) profiles, UML notation is used to notate a fundamentally different computational paradigm than the traditional OO programming language heritage of UML. The paradigm in question is one of components which communicate by passing messages or events, through channels connected to their input and output ports. Channels are generally asynchronous and can be used to broadcast the same message to many components. Although such a framework can be implemented on top of an OO programming language, the paradigm is conceptually very different to the basis of OOPLs. However, there is some commonality between the languages. For example, object-oriented structuring, as depicted by class diagrams, can also also be used to structure the information delivered by messages. It is this commonality that suggests it may be possible do define families of languages. Doing so, could considerably ease the burden of developing tools to support different languages.

It is barely necessary to make the point that different processes, involving different model architectures, will be required in different domains, often in different organisations, and sometimes in different projects. Indeed some have argued (e.g. [14]) that nearly every project needs the flexibility to "roll their own"; and in MDE this also means identifying the kind of models they are going to maintain, and the tools for developing and coordinating those models. As there is commonality between languages, there is also commonality between processes and it follows that it may be possible to define families of processes, with the desirable result of making it easier to develop/configure tools to different processes.

But this could make MDE (and MDA) untenable as a widespread approach. I have argued quite strongly that tools are essential to reduce the burden in maintaining many different, related models to a level that outweighs the benefits of abstraction that models can provide. Building tools by hand is resource intensive, and certainly could not be done for every process "rolled" for a project. Of course standard forms of MDE could be established for the development of particular kinds of system, by picking a particular process and particular kinds of models to be maintained, and then hand-building tools to support that package. It is unlikely that such 'solutions' would be very configurable and there is a great danger that developers would be forced into too much of a straightjacket.

An alternative is to work out how to generate and/or configure tools from definitions of processes and languages. This is the topic of the next and concluding section of this article.

8 Going Meta

Somehow we need a way to go "meta". We need to be able to feed language and mapping definitions into tools that will then either generate other tools or configure themselves according to that definition. This is, in a sense, applying MDE/MDA to itself. Language definitions are just models, possibly models with mappings defined between them (e.g. mapping of concrete notation(s) to abstract syntax). Is there a PIM language good for defining languages from which tools (PSMs) can be generated?

Technologies are beginning to emerge that are allowing this to happen in practice, for example the Extensible Markup Language (XML) and (more importantly, in my view) the OMG's Meta-Object Facility (MOF). XML is gaining widespread use in industry as a means of interchaging information that conforms to a particular structure. In other words, interchanging expressions of a precisely defined language. XML forces the information to be structured as a tree, though it does offer adhoc devices to break that structure when necessary, and its syntax is only just about human readable. On the other hand, there are plenty of tools to support XML, such as parsers, checkers (to check well-formedness) and transformation technology in the form of XSLT. Furthermore, these tools are meta-tools in the sense that they work with expressions of any XML-defined language, provided they have access to the definition of that language in the form of an XML-schema.

MOF is a language used for metamodelling. In general useage, this has come to mean the use of an object-oriented modelling language for the definition of languages. Typically the language used is MOF, or an appropriate subset of UML. All OMG standards are now defined in this way; in particular a goal in the development of MOF version 2 [6], is to make MOF *the* subset of UML used for metamodelling.

The advantage of using MOF to define languages is that, like, XML, MOF definitions are machine processable. Specifically, the MOF standard dictates how MOF models and instances of MOF models (languages and language expressions) may be rendered in XML format (schemas and XML documents, respectively), and how interfaces to repositories for models can be derived from MOF definitions of the languages in which those models are expressed. There is work in the pipeline to add a capability in MOF to express transformations between MOF models, that is language translations.

Metamodelling has only really been used to define the abstract syntax of a modelling language. However, there is recent work to show that concrete syntax and semantics can be defined using a metamodelling approach [3,8,4,19,18]. In this work, it is proposed that concrete notation is defined as another metamodel, as is the mapping to abstract syntax. Semantics is defined using a denotational approach, where a metamodel definition of the semantics domain (or instances of models) is provided, together with a metamodel definition of what it means for an instance to satisfy a model. If MOF is extended with a specialised language for expressing mappings, and this language is both rich enough and is supported by tools implementing those mappings, then there is a chance that some of the tools described in section 6, in particular those concerned with the relationship between instance and model, could be derived automatically from language definitions. Similarly, it should be possible to derive tools that transform concrete to abstract syntax an vice-versa.

One question here is whether the transformation language can be made expressive enough in a way that does not break the ability to derive tools from the definition. This is certainly an interesting topic for research, which could build on much of the work done in formal language theory. In particular, one promising source of inspiration could be graph grammars [10].

Another question concerns the richness of the semantics that can be expressed in this way. It would be very interesting to explore the relationship between standard, mathematical approaches to giving semantics and this approach. Could the metamodelling approach provide a pathway through which mathematical style semantics could be be used to generate automated or semi-automated analysis tools?

An area where MOF is weak is in its support for well-formedness rules. Although OCL is recommended as a (semi-)formal way of expressing these rules, MOF tools tend to ignore them. If well-formedness rules are to be checked and/or enforced then constraints need to find their way into repositories generated from MOF definitions. This should not be that difficult to do. For example, there are already Java libraries that allow OCL constraints to be monitored [12].

There is a clear need in the OMG for metamodelling techniques to support the definition of families of languages, but there are no explicit plans to properly support this within MOF. However, we have been working on this problem. Initial ideas are reported in [3,8], and these are being developed further in a submission to the UML 2 RFPs [5], latest versions of which are available from the 2U submission team's website [21].

With regard to defining processes, there is also a UML profile for defining software development processes, called the *Software Process Engineering Management (SPEM) UML Profile* [7]. This is an initial attempt at defining a language, which shares some of UML's constructs, for defining software engineering processes. The language is being implemented; for example, Rational's Unified Process tool is intended to be configurable with SPEM definitions. There is a considerable amount of work to do in integrating SPEM with MOF, and generating process coordination tools that integrate with modelling tools from SPEM definitions.

Acknowledgements

I would like to thank Tony Clark, Steve Cook, Andy Evans and Alan Wills for many useful discussions on ideas presented in this paper. This work was partially supported by a Royal Society Industry Fellowship.

References

1. Aspect oriented software design (AOSD) home page. http://www.aosd.net.
2. Catalog of OMG modeling specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm.
3. Clark A., Evans A., Kent S., Brodsky S., and Cook S. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modeling approach. Available from www.puml.org, September 2000.

4. J. M. Alvarez, A. Clark, A. Evans, and P. Sammut. An action semantics for MML. In C. Kobryn and M. Gogolla, editors, *Proceedings of The Fourth International Conference on the Unified Modeling Language (UML'2001)*, LNCS. Springer, 2000.
5. OMG Analysis and Design Task Force. UML 2.0 requests for proposals. Available from http://www.omg.org/techprocess/meetings/schedule/, 2001.
6. OMG Analysis and Design Task Force. MOF 2.0 requests for proposals. Available from http://www.omg.org/techprocess/meetings/schedule/, 2002.
7. OMG Analysis and Design Task Force. SPEM final adopted specification. OMG document number ptc/02-01-23, available from [2], 2002.
8. A. Clark, A. Evans, and S. Kent. Engineering modelling languages: A precise meta-modelling approach. In *Proceedings of ETAPS 02 FASE Conference*, LNCS. Springer, April 2002.
9. D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.
10. H. Ehrig, G. Engels, H-J. Kreowski, and G. Rozenberg, editors. *Handbook Of Graph Grammars And Computing By Graph Transformation. Volume 2: Applications, Languages and Tools*. World Scientific, October 1999.
11. W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 411–428. ACM, September 1993.
12. Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.
13. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 1997.
14. A. Lauder. *A Productive Response To Legacy System Petrification*. PhD thesis, Department of Computer Science, University of Kent, UK, January 2002.
15. B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specifications. *IEEE Transactions on Software Engineering*, 20(10):760773, October 1994.
16. OMG Architecture Board ORMSC. Model driven architecture (MDA). OMG document number ormsc/2001-07-01, available from www.omg.org, July 2001.
17. Eclipse Project. Home page. http://www.eclipse.org.
18. G. Reggio. Metamodelling behavioural aspects: the case of the UML state machines (complete version). Technical Report DISI-TR-02-3, DISI, Universit di Genova, Italy, 2001.
19. G. Reggio and E. Astesiano. A proposal of a dynamic core for UML metamodelling with MML. Technical Report DISI-TR-01-17, DISI, Universit di Genova, Italy, 2001.
20. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans and S. Kent, editors, *The Third International Conference on the Unified Modeling Language (UML'2000), York, UK, October 2-6. 2000, Proceedings*, LNCS. Springer, 2000.
21. 2U Submitters. Home page. http://www.2uworks.org.
22. P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton. N degrees of separation: Multidimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, May 1999.