Applying Model-Driven Development to Pervasive System Engineering

Huy N Pham and Qusay H Mahmoud
Department of Computing & Info Science
University of Guelph
Guelph, Ontario, Canada
{hpham, qmahmoud}@uoguelph.ca

Alexander Ferworn and Alireza Sadeghian
Department of Computer Science
Ryerson University
Toronto, Ontario, Canada
{aferworn, asadeghi}@ryerson.ca

Abstract – *This paper provides an overview of model-driven software development methodologies and argues that they are well-suited for the purpose of pervasive software development. It also surveys some of the most notable model-driven pervasive software development frameworks that are available, including one of our own.*

1. Introduction

People, for the most part, interact with their computers not because they enjoy to, but rather because they need to get things done. Perhaps it's some poetry that they want to write. Perhaps it's a financial report that they need to prepare. Or perhaps it's a paper that they need to edit. In most cases however, the ultimate purpose is to accomplish something else. Offer them a choice of accomplishing the same objectives by using a computer or, say – "taking a walk in the woods" [1] instead, and most people would opt for the latter. So, except for those who are intimately involved in system testing (or something of that nature), the computer is just another tool that people have to use, or interact with, in order to accomplish something else. With this observation in mind, it is not too difficult to see that one of the major problems of the current computing paradigm is that it is still very obstructive and unfriendly: Instead of fitting naturally into the human's environment, today's computers "*force the humans to enter theirs*", as noted in Mark Weiser's seminal paper [1].

Initiated a little more than a decade and a half ago based on Weiser's vision, Ubiquitous or Pervasive Computing (PC) is a vision that aims to make computing a technology that is user-friendlier and more human-centric, for – in Weiser's words – "*The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are in-distinguishable from it*". In this vision, our living and working environments will be saturated with computing and communication capabilities that are designed around humans instead of machines, and users will have access to computational services that are available all the time and everywhere, yet in a transparent and un-obstructive way [2].

With more than a decade of progress in hardware capabilities and advances in several key technologies in software and networking, Pervasive Computing is becoming a reality [2, 3], and there have been a large number of pervasive computing projects around the world that try to prove the viability and usefulness of this vision in a variety of domains. Unfortunately, along with the new and exiting possibilities come the difficult challenges. As pervasive applications are highly distributed and mobile in nature, pervasive system developers have to face all the challenges found in the fields of Distributed Systems (DS) and Mobile Computing (MC) [4], as wells as a new set of challenges that did not exist in either DS or MC alone [5]. As discussed in [3], PC system developers need to deal with issues like *scalability* – the ability of a pervasive system to handle large number of devices that could join and leave the system on a momentarily basis without warning, and the intense level of interaction between the system and its users; *heterogeneity* – the large variation in the execution and programming models of the devices in the system, and the uneven conditioning of communication and other resource infrastructures; and *invisibility* – the ability of the system to operate with minimal required human intervention, usually by means of adaptive behaviors, sophisticated system perception (i.e. context awareness) and intelligent context management, etc.

While computational perception and intelligent behaviors are integral and central to PC applications, heterogeneity and scalability are perhaps the most challenging factors of PC from a software engineering point of view. These two intrinsic characteristics of pervasive computing make software development in this domain a very challenging task. Firstly, hardware devices used in PC are often "*resource-poor*", and might have very limited memory, storage, battery, and computational power [4]. As a consequence, middleware solutions might be impossible in some cases [6], and multiple versions of the same software – one for each target execution platform – may need to be separately implemented and maintained. Secondly, the need to develop pervasive systems in an "*organic fashion*" [5], together with the fact that hardware and implementation technologies have tended to change quickly in this domain, imply the need for frequent updates to be performed on the system. This fact, compounded with the need to maintain multiple versions of the same software,

First International Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments (SEPCASE'07)
0-7695-2970-4/07 $20.00 © 2007 IEEE

makes software maintenance in this domain a potentially complex and costly process.

Model-Driven Development (MDD) [7] is a relatively new software engineering approach that promises to overcome the above difficulties and significantly increase the productivity of software development and maintenance. This paper provides an overview of MDD methodologies and their applications to the development of pervasive systems. The paper is organized as follows. Section 2 introduces the ideas behind MDD, its terminology, standards, and tooling. Section 3 describes the potential benefits of applying MDD to the development of pervasive systems. Section 4 describes some of the most notable MDD-based pervasive development projects. Finally, section 5 concludes the paper.

2. Model-Driven Development

Since the introduction of high-level programming languages like C, C++ and Java, software developers have seen a big breakthrough in productivity. Instead of having to construct their applications from primitives like *load*, *store* and *jump*, developers are able to express their logic using higher and more natural programming constructs like *loops*, *conditionals*, etc., and then call upon a compiler to generate (virtual) machine-level code. Also, using these languages, the task of porting software to a new execution platform when it comes along involves only a recompile, as opposed to a rewrite or a redesign, of the programs in the system. These benefits came from the basic idea that, by raising the abstraction level of the language used by developers, and by automating the process of transforming code from the raised abstraction level to the target (lower) abstraction level, productivity in both software development and maintenance can be significantly improved.

Model-Driven Development is a software engineering approach that aims to push this idea one step further. It proposes a software development methodology in which software is developed not by writing code directly in implementation languages, but by constructing high level models that can be transformed into code by automated transformation engines and code generators, as illustrated in Figure 1 below.

Figure 1. MDD software development

Like high-level languages, this approach offers three major advantages. The first is that, since high-level modeling concepts, as compared to those found in implementation languages, are much closer to the real concepts in the problem domain, software specification, understanding, and development are much easier with models. The second major advantage, which is captured in MDD's "Model once, generate anywhere" slogan, is that, since the concepts used in the models are less bound to the underlying implementation technology, software is less susceptible to technological change. This makes software maintenance easier and more economical. The third advantage is that, since expert implementation knowledge is encoded into the transformer, it can easily be reused and shared between different projects and teams, increasing both the productivity and quality of software development.

2.1. Terminology

Central to the MDD approach is the concepts of *models* and *modeling*. A model is a form of abstraction that allows real-world entities to be represented in a simplified manner, so that they can be dealt with in safer, cheaper and easier ways [8]. In other words, a model is a set of statements about something that is under study, and either describes that something, if it already exists, or specifies it, if it does not exist.

A *modeling language* is a language in which models are expressed. It gives the models their meanings, by mapping the concepts or entities in the models to the concepts or entities in the domain under study. Modeling languages are defined by their *syntaxes*, *semantics*, and *notations* [9]. A language's syntax -- also known as the abstract syntax -- describes the vocabulary of the concepts provided by the language, as well as how those concepts can be combined together to form models. The semantics describes the meanings of the concepts defined in the syntax, and the notation – also known as concrete syntax – describes how the abstract syntax is to be presented to the user of the models.

Like programming languages, modeling languages differ in their levels of generality. As a rule, a language's scope of application is proportional to its generality, while its expressive power in a given problem domain is proportional to its specificity with respect to that particular domain [10]. Languages that are specific to a certain problem domain are called *domain-specific languages*, or *DSLs*. It has been argued [11] that in order to be useful for the purpose of automated code generation, and hence MDD, modeling languages will have to 1) have well-defined syntaxes and semantics, and 2) be domain-specific. The first requirement gives models their well-defined meanings and makes them machine-interpretable. The second requirement ensures that the models can be expressive and powerful enough to allow real-world entities to be usefully represented. For this reason, there has been a growing trend in using DSLs, as opposed to general languages like the Unified Modeling Language (UML), as the modeling languages in MDD.

In order for a modeling language's syntax, semantic and notation to be specified, another language will be needed. This language, which may or may not be the same as the modeling language itself, is called a *meta language*. A meta language can be considered to be a tool that is used by language designers, or engineers, to design and specify modeling languages. This specification, created by the language designer using a meta-language, is called a *meta*

model – the model of a modeling language. Similarly, a *meta meta model* is a model that describes a meta language, and a *meta meta language* is the language in which meta meta models are expressed [12]. In principle, one can ascend the meta-levels indefinitely. In practice however, it has been generally agreed that three levels of modeling, with the meta meta language being its own specification language, is the most productive. In this setting, the models at the lowest level are called the M_1 models, while meta models are called M_2 and meta meta models are called M_3. Also, M_0 refers to the entities that are being modeled [13].

Once the software system has been completely specified using a set of models, the next step in the MDD approach is to transform those models into an actual implementation. This process, in the context of MDD, is done with the use of *model transformation engines*, which take a model as input, and produce as output either the corresponding implementation code or another intermediate model. In order to do this, the transformation engine needs to be provided with both the source and target meta models, as well as a set of mapping rules, or *model transformation rules*, which map entities and relationships in the source meta model to entities and relationships in the target metamodels. Also, a *transformation language* will be needed to express models transformation rules. Section 2.4 provides a list of notable model transformation languages that are currently available.

2.2. Pragmatic Issues

As automated software generation has been tried before with very limited success [14], in order for MDD approaches to be useful they will need to effectively deal with a variety of pragmatic issues. These issues can be divided into two broad classes. The first are those related to the construction of models, and the second are those related to their transformations. As mentioned earlier, MDD's models, first and foremost, need to be machine interpretable and have precise meanings. Also, it is desirable for models to have other qualities such as good abstraction mechanisms, easy to understand, inexpensive to construct, and be predictive (i.e., they allow the prediction of essential properties regarding the modeled system). Meeting these requirements are the primary objectives of the modeling language community.

While high quality modeling languages are essential to the specification of software systems, it has been said [15] that model transformation is the "heart and soul" of MDD. Given a set of quality models, it is the job of the model transformation engine(s) to produce an efficient implementation of the actual system. Among the most frequently mentioned pragmatic issues of model transformation are traceability – the ability to relate generated elements to their original elements, efficiency of generated code, scalability and integratability [16]. While current model transformation approaches have achieved reasonable results in the last four criteria [17], traceability, which is useful for model synchronization, impact analysis (i.e., analyzing how changes in one model would affects other related models), and model-based debugging, is still a major issue. [18] provides a taxonomy of different model transformation approaches, while [19] provides a taxonomy of model transformation design features, and describes how these features can be used to address the pragmatic issues mentioned above.

2.3. Prominent Methodologies

Two of the most prominent MDD methodologies are the Model Driven Architecture (MDA) [20] by the Object Management Group (OMG), and the Software Factories framework (SF) [11] by Microsoft Corporation. Both of these methodologies call for the treatment of models as the primary artifacts – as opposed to an overhead that consumes development resources – in the software development process. They differ however, on how general (or specific) those artifacts should be, and how they are to be transformed to produce the actual implementation of the system.

2.3.1. Model-Driven Architecture

MDA [20] is an initiative from the OMG with the objective of providing standards and guidelines that are useful for the practicing of MDD. Central to this architecture are two related concepts that represent its primary thesis, besides the call for treating models as the primary artifacts in the software development process, like other MDD approaches do. The first concept is *(implementation and) platform independence* – the absence of all information about the underlying implementation technologies and execution platform in a software model. The second concept is *separation of concerns* – the separation between a software system's business concerns (which determines *what* the system is supposed to provide) and its technological or implementation concerns (which determines *how* it is going to provide them). More specifically, it is MDA's primary argument that, in order to make software maintenance and evolution more economical, all business concerns of a software system should be captured using a platform independent model (PIM), while its technological concerns should be separately captured inside the transformation engine that translates the PIM into lower level models. This way, as the system's implementation technologies or execution platform change, the only thing that needs to be updated is the transformation engine, and all development efforts invested in the construction of its business logic (i.e., the PIM) can be saved.

To promote this practice, MDA proposes a three-level architecture, as illustrated in Figure 2 below.

Figure 2. MDA's three-levels architecture

0-7695-2970-4/07 $20.00 © 2007 IEEE

As can be seen, at the top level is the PIM. This is the model of the system's business logic, and is to be manually constructed by the system designer from its requirement specifications. As its name implies, this model should describe what functionalities and features the system should provide without mentioning anything about its underlying implementation technologies or execution platform. At the second level are the Platform Specific Models (PSMs). These models are to be automatically generated from the PIM by the transformation engine, and represent the possible implementations of the system (i.e., there should be one PSM for each target implementation technology and execution platform). At the lowest level are source code files in some implementation languages (e.g., C#/Java or some middleware languages). These files are to be generated from the PSMs by the code generator and can be compiled by a compiler to produce the executable implementation for the system.

As officially defined by the OMG, the PIM must be a UML model. Because UML is not a precise language [11] however, there are other MDA interpretations in which the PIM is to be constructed using some other modeling language. In this relaxed interpretation, a software development framework needs to provide the following building blocks in order to implement this architecture: 1) A (domain-specific) modeling language for constructing the PIM, if it is not going to be UML; 2) One PSM language for each target implementation platform; 3) A transformation mechanism to translate the PIM into the PSMs (a transformation engine); 4) A transformation mechanism to translate the PSMs to implementation code (a code generator).

One of the most frequently cited weaknesses of MDA is that, while the PSMs to code transformation is relatively straightforward, because PSM languages are supposed to have a close correlation with their corresponding implementation languages, the transformation between the PIM and PSMs usually requires jumping a wide abstraction gap, and is very difficult to perform efficiently in one single transformation. In order to be successful, great care need to be taken to ensure that all related pragmatics issues can be adequately addressed [21]. In [22], a more agile approach to MDA that incorporates a series of verification steps into the process to make it more robust and practical is described.

2.3.2. Software Factories

Software Factories (SF), proposed by Microsoft, is another well-known MDD methodology. Despite the fact that SF is officially defined as Software Product Lines [23] and that its primary thesis lies in that area, this framework is also based on the principles of MDD, and therefore exhibits all the essential characteristics of an MDD methodology.

In [11], SF's architecture is described using the concepts of *viewpoints* – well-defined perspective from which a given aspect of a software product can be described and specified [24], *views* – an instance of a viewpoint, and *(software factory) schema* – a group of related viewpoints arranged in a grid-like form that allows development artifacts and assets to be organized and managed in an orderly way. A software factory is then defined as an SF schema populated with the necessary software development assets. That is, to build a software factory, software factory developers need to define an appropriate SF schema, and populate each of its viewpoints with software development assets such as DSL (to allow models of that viewpoint to be constructed), transformation engines (to allow models of that viewpoint to be transformed to models or codes of adjacent viewpoints), etc. Once the SF schema has been populated (at which point it becomes an *SF template* – an instance of the schema), the DSLs from appropriate viewpoints can be used to model the different aspects of a software system, and transformation engines between the viewpoints can be used to produce the actual implementation for that system.

As can be seen, SF is different from MDA in two important aspects. The first is that it does not aim for complete platform independence, and the second is that it proposes the use of multi-step progressive model transformations. These two differences reflect SF's strategy for addressing the practicality issues that are encountered by MDA. By using multiple DSLs to specify the system at various levels of abstraction (i.e., viewpoints), SF gives up platform independence to allow a more detailed and precise models of the system to be constructed. Also, by performing multiple incremental transformations between these abstraction levels, as oppose to a single PIM to PSM transformation in the case of MDA, SF aims to make the transformation process more concrete and tractable, and produce a more efficient implementation.

As mentioned above, SF is more than just an MDD approach. It fact, the primary debut of SF is that, by building a software factory, domain experts (i.e., people who have developed software systems for a certain domain on multiple occasions in the past) can communicate their know-how knowledge (in the form of processes, architectures, transformation mappings, patterns, etc.) and tooling (in the form of DSL, transformation engines, etc.) to other software developers, allowing them to rapidly build quality software in the domain in an economical and efficient way. As such, it can also be a valuable tool for the pervasive system developer, as described in section 4 below.

2.4. Tooling

While they provide the overall guidelines and architectures for doing model-driven development, both MDA and SF do not provide all the concrete techniques and facilities required to implement their proposed strategies. In the case of MDA, besides UML, which was originally proposed as the modeling language for PIMs, there is also a meta modeling standard called *Meta Object Facility* (MOF) [25], which specifies a common architecture to which all meta modeling tools should adhere to ensure

interoperability between tools, a (soon to be finalized) model transformation standard called *Queries/Views /Transformations* (QVT) [26], which attempts to standardize the way model transformations are to be done in MDA. In the case of SF, there is a meta modeling and (simple) code generation tool called DSL Tools [27]. For the remaining components, MDD developers would have to rely on third-party tools. While there are a large number of choices for modeling tools, there are relatively much fewer choices for metamodeling (i.e., language specification) and model transformation tools. Table 1 and 2 list some of the most popular third-party metamodeling and transformation tools/languages that are currently available.

Table 1. Some popular metamodeling tools

Name	Vendor	License Type
MetaEdit+	MetaCase	Commercial
XMF-Mosiac	Xactium	Commercial
KerMeta	Kermeta.org	Open source

Table 2. Some popular model transformation tools

Name	Vendor / Institution	License Type
Mia Transformation	Mia Softwares	Commercial
PathMATE	Pathfinder Solutions	Commercial
Atlas Transformation Language (ATL)	INRIA	GPL
QVTEclipse	QVT-Partners	GPL

3. Notable MDD Pervasive Projects

One notable MDD-based pervasive proposal is an MDA-compliant pervasive software development framework described in [28]. This framework allows pervasive software developers to create OSGi-based (an application middleware platform [29]) pervasive applications by constructing platform independent models in a UML-based domain-specific language called PervML. Figure 3 below shows the major components of this framework.

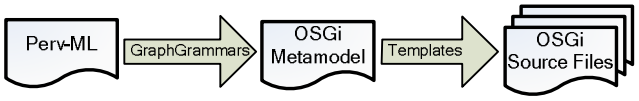

Figure 3. Munoz, et al.'s MDA implementation

As can be seen in the figure, this framework implements the MDA architecture by providing a modeling language (PervML) for the construction of PIMs, a language (OSGi Metamodel) in which the PSM can be expressed, a PIM-to-PSM model transformation mechanism (based on graph grammars), and a code generation mechanism (based on text templating). While it still remains to be seen how the framework as a whole can be applied to the development of real-world pervasive projects, the two modeling languages, PervML and OSGi metamodel, can be considered to be the proposal's initial contributions. Perv-ML is a UML-base graphical language that allows different kinds of model, each kind reflects a different view of the system analyst and architect, to be constructed, and can be useful for the purpose of pervasive system modeling. OSGi metamodel is produced by abstracting the OSGi middleware language, can be used by other MDD projects that use OSGi as the target execution platform.

Another notable MDD-based pervasive project is a Lego-like application development framework called VRDK, described in [6]. This framework allows pervasive software developers to construct their applications by dragging graphical modeling primitives representing well-defined hardware or software components into a graphical design surface and then connecting them together. Once the model has been constructed, the developer can either simulate the designed application using the framework-supplied model interpreter, or generate and deploy the actual application using model transformer. The set of graphical modeling primitives is extensible by mean of a plug-in facility that allow user to define new system components. Like the previous framework, it remains to be seen how this framework can be applied to the development of real-world pervasive projects.

Another MDD pervasive development project that we would like to describe is a project in which we applied MDD to the development of an Search and Rescue (SAR) support system. This project, called Canine Augmentation Technology (CAT) [30], resembles a pervasive system in many aspects and is part of a cooperative effort between our group and the Provincial Emergency Response Team of the Ontario Provincial Police. Its primary intent is to provide useful technological component augmentation to the force's USAR Canine teams in support of SAR operations where direct interaction is precluded, including those that involve searching rubbles, partially collapsed and unsafe structures. More specifically, CAT's main purpose is to provide the force's emergency canine handlers with the ability to see what the dogs see, hear what the dogs hear, and communicate with their dogs during SAR operations. From an engineering perspective, CAT consists of three different sub-systems, one to be mounted on the dogs, one to be carried by the dog handlers, and one to be used by the central observer/coordinator who coordinates the whole searching effort. The purpose of the dog's system is to gather, encode, and transmit sensed data such as the dog's GPS location, video and audio, etc., to both the handler and observers. The purpose of the handler's system is to receive data from the dog, gather and transmit the handler's location data to the observer, and transmit the handler's voice commands to the dog. Finally, the observer system gathers all data transmitted by all the dogs and theirs handlers, and presents them to the observer in a coherent manner.

During the course of the project, we have come to notice that not only many of the software components, such as those that capture, transmit and process GPS, video and audio signals, are reusable between different SAR support systems, the designs and good practices used in these components, such as how the signal transmitters and receivers can be efficiently implemented, can also be encoded in some forms of a framework. This framework,

we believed, could help increase both the productivity and the quality of similar projects because it would allow systematic reuse of both the components and the best practices. So we set out to develop such a framework, and the result is a (domain-specific) modeling language that encapsulates the common building blocks of SAR applications as pre-defined graphical modeling primitives. The meta modeling tool used was Microsoft's DSL Tools. Figure 4 below shows our graphical modeling environment, hosted inside Microsoft Visual Studio.

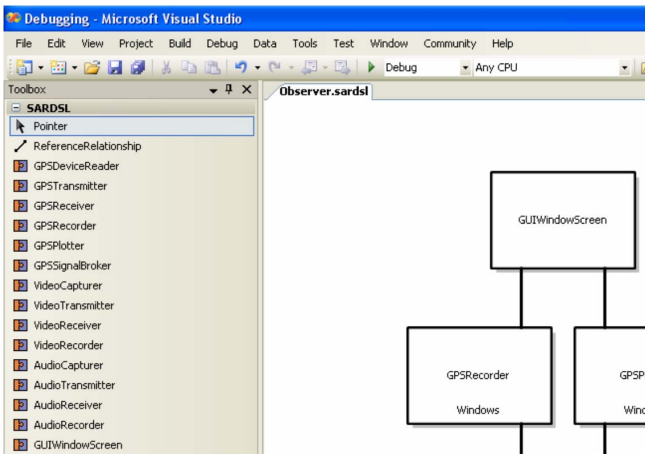

Figure 4. A SAR software development environment

Using this development environment, SAR software developers can quickly compose and generate their applications by dragging the graphical primitives shown in the toolbox on the left, which represent common SAR software components, onto the design surface, configure them by specifying the appropriate parameters, and then use our supplied text generation templates to generate the application codes.

We have used our DSL in the development of the GPS, audio and video components of CAT. We are also planning to extend the language to include a more comprehensive set of software building blocks, and apply it to different development SAR software development projects to test it applicability and efficiency.

4. Benefits of Applying MDD to Pervasive System Development

As pervasive software projects are software engineering projects in which frequent updates are necessary, their development, maintenance and evolution can greatly benefit from the application of MDD techniques. The set of potential benefits includes:

- By constructing the system using abstract models rather than low-level code, system maintenance, update and evolution will be much more productive and economical, especially in the context of pervasive computing, where technologies tend to change very rapidly, and frequent, incremental updates are necessary.
- Being able to work at a higher level of abstraction also allows pervasive software developer to deal with heterogeneity much more easily.
- Simulation and early requirement validation is much easier with the availability of a model. This feature is especially helpful for development projects that are highly hardware dependent, such as pervasive computing projects, because the hardware platforms on which the software being developed will run might not be available during the early stages of the development process.
- Because system design and specification is much more intuitive using models than with other kinds of development artifact, developers of pervasive systems will be in a better position to deal with complexity.
- MDD facilitates systematic reuse of know-how knowledge, software best practices and development assets, and can be especially useful for newly established fields like pervasive development.

In our CAT project, we were not able to take advantage of the first two benefits above. The reason is that, with CAT being the first and only "customer" of our DSL, there is not much differences in the amount of effort required to modify or extend the software components involved had we implemented CAT without using the DSL. We were not able to take advantage of the third benefit either, because our DSL does not have the capabilities required for model execution (i.e., simulation). We were able to take advantage of the last two benefits, however. High-level modeling primitives made it much easier to explain the software's design to new team members, for example, and the good practices and experiences that we learn during the project can be encoded and retained inside the code generation templates.

5. Conclusion

Eager to prove the feasibility of Weiser's vision, today's pervasive environments and applications, for the most part, were developed in an *ad hoc* manner. As a result, pervasive computing is still "*more of an art than a science*" [31]. As most software systems envisioned by the pervasive computing vision are expected to be large and complex, current software engineering methodologies will not be able to deal with their level of complexity. In this paper we have provided an overview of Model-Driven Development methodologies, and argued for the use of these methodologies to deal with the complexity and challenges, such as scalability and heterogeneity, of pervasive systems.

References

[1] M. Weiser, "The Computer for the 21 Century," in *Scientific American*. vol. 256, 1991, pp. 94-104.

[2] M. Satyanarayanan, "Pervasive computing: vision and challenges," in *Personal Communications, IEEE Computer*. vol. 8, 2001, pp. 10-17.

0-7695-2970-4/07 $20.00 © 2007 IEEE

[3] D. Saha and A. Mukherjee, "Pervasive computing: a paradigm for the 21st century," in *IEEE Computer Society*. vol. 36: IEEE, 2003, pp. 25-31.
[4] M. Satyanarayanan, "Fundamental challenges in mobile computing," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996.
[5] G. D. Abowd, "Software engineering issues for ubiquitous computing," in *Proceedings of the 1999 Intl. Conference on Software Engineering*, Los Angeles, CA, USA, 1999, pp. 75-84.
[6] A. Ulbrich, T. Weis, K. Geihs, and W. Allee, "A Modeling Language for Applications in Pervasive Computing Environments," in *Proc. of the 2nd Intl. Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2005.
[7] M. Volter and T. Stahl, *Model-Driven Software Development: Technology, Engineering, Management*: John Wiley and Sons Ltd, 2006.
[8] J. Rothenberg, *The nature of modeling*: John Wiley & Sons, Inc. New York, NY, USA, 1989.
[9] T. Clark, A. Evans, P. Sammut, and J. Willans, "Applied Metamodelling: A Foundation for Language Driven Development," Xactium Co. Available at http://www.xactium.com/, 2004.
[10] K. Czarnecki, "Overview of Generative Software Development," in *Unconventional Programming Paradigms*, 2005.
[11] J. Greenfield and K. Short, *Software factories: assembling applications with patterns, models, frameworks and tools*: John Wiley and Sons, Ltd., 2004.
[12] C. Atkinson and T. Kuhne, "Model-driven development: a metamodeling foundation," *IEEE Software,* vol. 20, pp. 36-41, 2003.
[13] J. Bézivin, "On the unification power of models," *Software and Systems Modeling,* vol. 4, pp. 171-188, 2005.
[14] B. Boehm, "A view of 20th and 21st century software engineering," in *International Conference on Software Engineering*, 2006, pp. 12-29.
[15] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *IEEE Software,* vol. 20, pp. 42-45, 2003.
[16] B. Selic, "The Pragmatics of Model-Driven Development," *IEEE Software,* vol. 20, pp. 19-25, 2003.
[17] D. S. Frankel, *MDA: Applying MDA to Enterprise Computing*: Wiley, 2003.
[18] T. Mens, K. Czarnecki, and P. Van Gorp, "A taxonomy of model transformations," in *International Workshop on Graph and Model Transformation*. vol. 4101, pp. 2005-02.
[19] K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," in *OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, 2003.
[20] A. G. Kleppe, J. B. Warmer, W. Bast, and A. Watson, *MDA Explained: The Model Driven Architecture: Practice and Promise*: Addison-Wesley Professional, 2003.
[21] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, "Transformation: The Missing Link of MDA," *Lecture Notes in Computer Science,* vol. 2505, 2002.
[22] S. J. Mellor, "Agile MDA," *MDA Journal,* 2004.
[23] P. Clements and L. Northrop, *Software product lines*: Addison-Wesley Boston, 2002.
[24] M. W. Maier, D. Emery, and R. Hilliard, "Software Architecture: Introducing IEEE Standard 1471," *IEEE Computer,* vol. 34, pp. 107-109, 2001.
[25] ObjectManagementGroup, "Meta Object Facility Specification," *OMG Document 00-04-03,* 2000.
[26] ObjectManagementGroup, "MOF QVT Final Adopted Specification," *OMG Document 05-11-01,* 2005.
[27] MicrosoftDevelopersNetwork, "Domain-Specific Language Tools," Microsoft. Available at "msdn.microsoft.com/vstudio/DSLTools/", 2005.
[28] J. Muñoz, V. Pelechano, and J. Fons, "Model Driven Development of Pervasive Systems," in *Intl. Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, Hamilton, Canada, 2004, pp. 3-14.
[29] TheOpenServicesGatewayInitiative, "OSGi Service Platform Release 3. Available at http://www.osgi.org/resources/spec_download.asp," 2003.
[30] A. Ferworn, A. Sadeghian, K. Barnum, H. Rhanama, H. Pham, C. Erickson, D. Ostrom, and L. Dell'Agnese, "Urban Search and Rescue with Canine Augmentation Technology," in *IEEE System of Systems Engineering (SoSE06)* Los Angeles, CA, 2006.
[31] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski, "Challenges: An Application Model for Pervasive Computing," in *International conference on Mobile computing and networking*: ACM Press, NY, USA, 2000.