Supporting Situational Method Engineering with ISO/IEC 24744 and the Work Product Pool Approach

Cesar Gonzalez-Perez
European Software Institute
cesargon@verdewek.com
http://www.verdewek.com/work

Abstract. The advantages of situational method engineering (SME) as an approach to the development, specification and application of methods are significant. However, taking this approach into practice in real-world settings is often a daunting task, because the necessary infrastructure and superstructure are not currently available. By infrastructure, we mean the underpinning theoretical and technological foundations on which SME is based; in this regard, this paper explains how the ISO/IEC 24744 metamodel solves many long-standing problems in methodology specification and enactment that other approaches, such as OMG's SPEM, cannot. By superstructure, we mean the exploitation mechanisms, often in the form of tools and decision procedures, that allow individuals and organisations to obtain value out of SME during their daily activities. Without these, SME is often seen as a purely theoretical exercise with little practical purpose. In this regard, we this paper also introduces the work product pool approach, which departs from the conventional view that methodologies must be described in a process-centric fashion to focus on a product-centric worldview, thus providing teams the capability to adopt an opportunistic and people-oriented setting in which to conduct their work.

1 Introduction

The method engineering approach [3, 13] builds upon the assumption that no specific methodology can solve enough problems and, therefore, methodologies must be specifically created for a particular set of requirements. In order to make this feasible and cost-effective, the old principles of modularity and reuse are utilised, and methodologies are said to be assembled from pre-existing method components, rather than created from scratch. Method components, consequently, take a very

Please use the following format when citing this chapter:

Gonzalez-Perez, C., 2007, in IFIP International Federation for Information Processing, Volume 244, Situational Method Engineering: Fundamentals and Experiences, eds. Ralyté, J., Brinkkemper, S., Henderson-Sellers B., (Boston Springer), pp. 7-18.

preeminent role in method engineering, since they comprise the raw material from which methodologies are obtained. Method components are often said to be stored into a repository, and the kinds of method components, as well as the relationships that are possible between these kinds, are given by an underpinning metamodel. Several metamodels have been proposed, such as OMG's SPEM 1.1 [16], SPEM 2.0 [18] (still under development) and ISO/IEC 24744 [12]. Of these, the latter is especially oriented towards method engineering, providing specific support for extant issues that other proposals, such as the ongoing versions of OMG's SPEM, have not been able to solve. This support occurs at two levels: on the one hand, the appropriate theory is established, so that a viable method engineering-based solution can be developed on top of it. This involves issues such as the interactions between the product and process sides of a methodology, or the specification of the endeavour domain from the metamodel domain. On the other hand, the necessary exploitation mechanisms are developed, so that an ISO/IEC 24744-based methodological solution can be used in practice to solve real problems, going beyond a mere academic exercise.

The next section briefly introduces the ISO/IEC 24744 standard metamodel. Section 3 focuses on infrastructural issues, describing the major theoretical aspects that are solved by the ISO/IEC 24744 standard metamodel, and explaining how they are relevant for the method engineering approach. Section 4, on the other hand, focuses on superstructural issues, describing how the work product pool approach works on top of repositories and methodologies in order to let software developers achieve their goal, i.e. deliver working software.

2 The ISO/IEC 24744 Standard Metamodel

ISO/IEC 24744 is an International Standard that defines a metamodel for development methodologies. Although it is geared towards *software* development methodologies, there is nothing in it that can prevent it from being applied to systems development methodologies or even other areas.

In this context, a *metamodel* means a semi-formal language capable of describing methodologies, and that these methodologies are models themselves. This is similar to what other metamodels (such as OMG's SPEM) claim to do, but with a larger scope. The ISO/IEC 24744 metamodel covers the following domain areas:

- Work units, also known as the process aspect of methodologies. This describes the work that has to be done in order to obtain the system to be delivered. SPEM and other metamodels also cover this area.
- Work products, also known as the product aspect of methodologies. This describes the artefacts that must be used and/or created in order to obtain the system to be delivered. SPEM and other metamodels also cover this area, although at a very high level of abstraction.
- Producers, also known as the people aspect of methodologies. This describes the roles, teams and tools that actually perform the work units and create or use the work products mentioned above. SPEM and other metamodels barely cover this area.

- Stages, also known as the temporal aspect of methodologies. This describes how work units, work products and producers relate to each other over time, providing a macro-structure for the methodology (and, consequently, to endeavours). SPEM and other metamodels often mix this area together with work units, using the same class in the metamodel to specify the "what" and the "when". This poses heavy limitations on the modularisation of methodologies, which, arguably should be avoided in a method engineering context.
- Model units, also known as the modelling aspect of methodologies. This describes the modelling building blocks that developers can use in order to construct the work products mentioned above. SPEM and other metamodels do not cover this area, assuming that UML or other modelling language will be adopted and magically made to work with the methodology.

The following sections describe the details of some of the particularities of ISO/IEC 24744 and how they make it especially appropriate for method engineering.

3 Theoretical Aspects

The theory underpinning ISO/IEC 24744 departs from the classic views implemented by other metamodels in some aspects, but still conforms to a very conventional object-oriented worldview. The following sections describe the details of this theory, focussing on how it can provide the infrastructure for the implementation of a method engineering solution.

3.1 The Strict Metamodelling Paradigm

According to the OMG's worldview, models represent their subjects strictly by means of "instance-of" relationships. In other words, a subject is an "instance-of" the entity that models it. No "instance-of" relationships may occur other than these. Because of this, elements organise themselves into layers, sometimes called "metalevels" in the literature, connected only by "instance-of" relationships. This is often depicted as the widely know stack of metalevels, usually labelled M0, M1, etc. This worldview is called the strict metamodelling paradigm, and, although prevalent within the OMG's technology suite, it has been widely criticised from academia [2, 7]. To the best of our knowledge, no convincing reasons have been shown to exist as of why the strict metamodelling paradigm should be accepted. On the contrary, it is usually presented as an *a priori* statement that is to be obeyed without further explanation.

ISO/IEC 24744 departs from this stance, and organises elements according to the communities of people that are involved in their production and usage (Figure 1). On the one hand, method engineers maintain repositories of method components and may use them to create methodologies, which, in turn, are used by software developers to create software products. Method engineers and software developers are two different communities that establish the boundaries between three different domains: the metamodelling domain, the methodology domain, and the endeavour domain. Each domain is a representation of the domain "below" it, in the sense that

methodologies represent endeavours and metamodels represent methodologies [5]. The concept of *representation* has been explored in the software engineering literature (e.g. [9, 20]), and goes beyond that of "instance-of". A full discussion of the concept of representation is out of scope of this paper; please see [7] for an extended treatment.

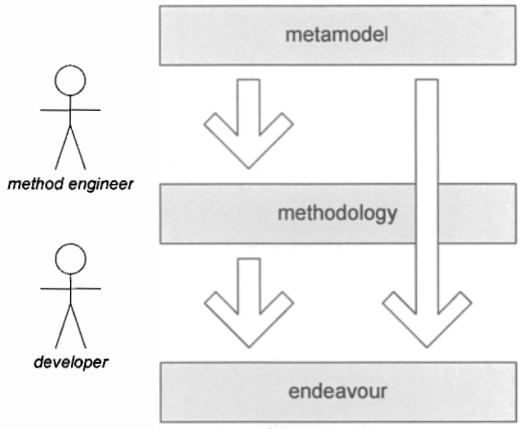

Figure 1. Overall structure of ISO/IEC 24744. Boxes depict domains, and arrows depict the representation relationships between domains. Stick figures depict the communities that are directly related to each domain or relationship between domains, as described in the main text.

The absence of the artificial restrictions imposed by the strict metamodelling paradigm, together with the flexible yet grounded concept of representation, gives ISO/IEC 24744 some capabilities that are discusses in the following sections.

3.2 Dual-Layer Modelling

The most important consequence of using the concept of representation as a means to relate domains, rather than that of "instance-of", is that the metamodel domain can exert control over multiple domains at the same time. With a traditional, strict metamodelling-based approach, a methodology is seen as an instance of a metamodel, and an endeavour as an instance of a methodology; therefore, there is no way in which a metamodel and an endeavour can be directly related. In other words, the designers of a metamodel cannot put in the metamodel anything that regulates how the endeavours that will be generated from the methodologies that will be obtained from the metamodel being designed will look like. For example, let us consider that the designers of a metamodel want to capture the fact that all the work products created during the application of any methodology must have a version number. Using a strict metamodelling approach, this is impossible, because work products exist in the endeavour domain, which is an instance of the methodology domain; therefore, the WorkProduct class, with its VersionNumber attribute, belongs to the methodology domain. The metamodel designers are free to design the metamodel as they wish, but cannot dictate anything about the methodology domain.

Methodologies are supposed to be put together by method engineers, not metamodel designers. Therefore, the metamodel designer cannot guarantee that all the work products created during the application of any methodology derived from the metamodel being designed will have a version number.

Using a representation-based approach, the chain of reasoning that led us to conclude that the WorkProduct class belongs to the methodology domain does not need to happen. On the contrary, a community-oriented perspective is taken. ISO/IEC 24744 assumes that any conceivable methodology will use work products, and therefore the concept of a work product is universal enough as to be "frozen" as part of the metamodel. In other words, the WorkProduct concept is provided to the method engineering community as raw material from which they can construct method components and populate repositories. The WorkProduct class, with its VersionNumber attribute, belongs to the metamodel domain. Its instances (i.e. specific work products, such as the requirements specification document that I can see on my desk as I type this) still belong to the endeavour domain. We must realise that this means that the representation relationship that links the WorkProduct class and its instances travels across the methodology domain; a class in the metamodel is being instantiated in the endeavour. This would be illegal in a strict metamodelling environment, but is perfectly reasonable in ISO/IEC 24744. The result is that the ISO/IEC 24744 metamodel is perfectly capable to exert control on the endeavour domain (e.g. determine that every work product will have a version number) as well as the methodology domain (see Figure 1).

Common sense dictates that the final purpose of any software development methodology is to produce working software. Therefore, any approach for the specification of methodologies should take into account the enactment (or application) of methodologies onto specific endeavours. Using a programming simile, we can say that an approach to methodology specification that does not take into account their enactment is akin to a programming language that can express programs but does not take into account the possibility of running them. The ability for a metamodel to provide classes that get instantiated at the endeavour level is not a plus, but something that should be essential. Furthermore, tracing between endeavour-level elements and methodology-level elements (method components) should be directly addressed by the structure of the metamodel. ISO/IEC 24744 achieves this by pairing classes that represent endeavour-level elements and methodology-level elements into *powertype patterns* [6], in which the methodology-level class (the powertype) partitions the endeavour-level class (the partitioned type). For example, ISO/IEC 24744 includes the classes Task and TaskKind. Task represents an actual task as performed at the endeavour level. TaskKind, on the other hand, represents a kind of task as documented in a methodology. Task has attributes such as StartTime or Duration. TaskKind has attributes such as Name or Purpose. Evidently, every task "is-of" a particular task kind. This is shown in the metamodel by pairing Task and TaskKind into the powertype pattern Task/*Kind, meaning that the TaskKind class (the powertype) partitions the Task class (the partitioned type) (

Figure 2).

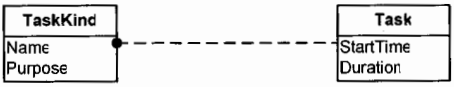

Figure 2. Powertype pattern formed by the Task and TaskKind classes in ISO/IEC 24744.

Since most metamodel classes are paired into powertype patterns, they are used together as well: a powertype pattern is "instantiated" as a whole. In fact, method components are created in ISO/IEC 24744 as clabjects [1], i.e. dual-faceted entities that exhibit a class facet plus an object facet. The object facet is obtained as a conventional instance of the methodology-level (powertype) class in the powertype pattern, whereas the class facet is obtained as a conventional subtype of the endeavour-domain class in the powertype pattern. Within a method component clabject, both facets, class and object, represent exactly the same concept, but using different representational mechanisms.

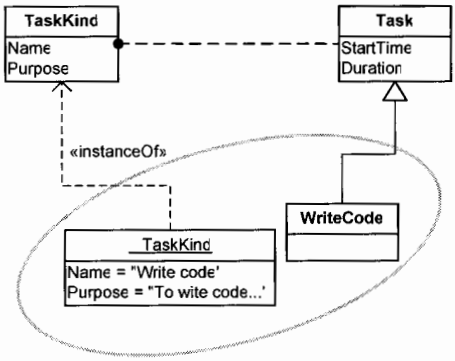

Figure 3. The "instantiation" of a powertype pattern. A regular object is instantiated from the TaskKind class, and a regular class is obtained by subtyping the Task class. Both together form a clabject (depicted by the ellipse), which is the implementation of a method component.

For example, if we were to "instantiate" the Task/*Kind powertype pattern to define a "Write code" task method component, this would involve creating an object as instance of TaskKind and giving values to its attributes (Name="Write code" and Purpose="To write code..."); and then creating a class (named WriteCode, for example) as subtype of Task, which would inherit its attributes (StartTime and Duration). The object with Name="Write code" and the WriteCode class (components of the clabject) represent the same concept, i.e. the task specification of writing code (Figure 3). The class facet will be useful as a template from which instantiation is possible during enactment, while the object facet is useful as "data" at the methodology level.

3.3 Product/Process Interaction

Another aspect that is often neglected by metamodelling approaches is that of the integration between the product and process aspects of methodologies. If we still

agree that the final purpose of any software development methodology is to produce working software, it should be clear that, at least from a motivational standpoint, methodologies should be product- rather than process-driven. To put it unceremoniously, process is a necessary evil. Process is necessary to transform ambiguous, incomplete and conflicting expectations given by stakeholders into hopefully working software. The ultimate purpose of a methodology, however, is obtaining the software, not performing the process. Interestingly, most metamodels for development methodologies are strongly process-focussed, sometimes even being called simply "processes" (hence the "P" in "SPEM"). The product aspect is usually assumed to be solved externally by the adoption of an all-encompassing modelling language such as UML [17, 19]. Such inattention to product results in a poor integration between process and product, because, given that the product aspect of the methodology to be used is unknown to the metamodel, very few assumptions about it can be made and, therefore, the interfaces between process and product elements in the metamodel must be kept to a minimum.

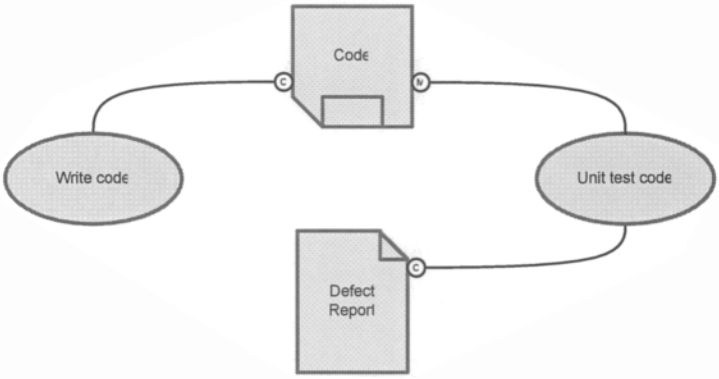

Figure 4. Action diagram showing that the "Write code" task kind creates work products of the "Code" and "Defect Report" kinds, and that the "Unit test code" task kind modifies work products of the "Code" kind and creates work products of the "Defect Report" kind.

ISO/IEC 24744 takes a different route, namely, that of modelling not only the process aspect but also the product aspect of methodologies. The former involves everything related to the tasks, processes, techniques and stages that describe the work- and time-oriented nature of the methodology, whereas the latter is concerned with everything related to the models, documents, languages, notations and model units from which models can be built. In other words, the product aspect of a methodology describes, with very fine details, the modelling languages (plus their potential forms of depiction) that may be used within an endeavour that follows the methodology. The whole of UML, for example, would be subsumed as part of a methodology defined as an instance of ISO/IEC 24744 (Figure 4).The major advantage of this approach is that, since the metamodel includes classes that represent both process and product aspects of the methodology, it can also specify rich interfaces between them. Conventional, process-focussed metamodels such as SPEM use an input/output approach to relate work products to the tasks that interact

with them. For example, using SPEM, we can say that the work product "Code" is an output of task "Write code", and an input to task "Unit test code". The input/output approach, however, is not rich enough to describe the variety of ways in which tasks can act upon work products. Consider the above mentioned task "Unit test code". When a developer performs that task, she starts with some code, unit-tests it, and then ends up with some code. Is the code at the end the same work product as the code at the beginning? With SPEM, we can only say, at the methodology level, that the work product "Code" is an input to "Unit test code" and also an output; it is not possible to reflect whether the outgoing product is a new instance of the Code work product kind, or the same instance that came in. ISO/IEC 24744 uses a richer approach, based on the concept of actions. An action is a usage event that a task performs upon a work product. An action is of a specific type: create, read-only, modify or delete. We can express a similar meaning to what in SPEM is conveyed by saying that the work product "Code" is an output of task "Write code" by saying in ISO/IEC 24744 that the task "Write code" is related to the work product "Code" via a "create" action. Notice that this meaning is richer, since we are specifying that the code is being created by the task rather than just put out. The unit testing example is more illustrative. With ISO/IEC 24744, we have the alternative of saying that the task "Unit test code" is related to the work product "Code" via a "modify" action, which means that the code that "comes in" and the code that "comes out" of the task, to maintain the input/output metaphor, is the same entity, but gets changed by the task. But we have the alternative to say that the task "Unit test code" is related to the work product "Code" via two actions: one of them is "read-only", and another one is "create". This means that unit-testing reads a piece of code but does not change it, and then creates a new piece of code as a result. From a methodological perspective, these two alternatives are very dissimilar, and can have dramatically different traceability, dependency and efficiency effects when the methodology is carried out.

4 Methodology Exploitation

Method engineering assumes that methodologies are assembled from method components and made available to their users, i.e. software developers. In order for software developers to reap the benefits of using these methodologies, the appropriate tools and techniques have to be provided as well. The following sections explore how software developers can exploit a methodology by using an alternative enactment approach and the work product pool approach.

4.1 Enactment Approaches

As we have said, most metamodels for development methodologies are strongly process-focussed. This is often reflected in the adoption of the metaphor that the organisation is a machine that executes the methodology as if it were a computer programme. Consequently, metamodels such as OMG's SPEM describe processes in terms of work breakdown structures and task precedence, borrowing from the most classical management styles.

This clashes frontally with the current trend towards agility [4, 22]. It is our personal experience that organisations often take advantage of nudging their systems, acting opportunistically when possible, and leveraging unexpected circumstances, even when an agile style of work is not explicitly encouraged, and more so when it is. Conventional management practices, on the contrary, try to impose up-front plans, which, from a methodological perspective, are sometimes seen as the one-off enactment of method components at the beginning of the endeavour. In other words, if a project plan is supposed to specify what is going to happen in terms of actual tasks and work products, why can't we instantiate the appropriate method components at the beginning of the project in a large big-bang enactment event and leave them sitting there for later use? Changes to the plan are supposed to be minor (as compared to the overall magnitude to the plan) and can surely be addressed by destroying or creating a few extra objects.

Let us consider, for a moment, that an up-front project plan is not possible or not wanted. The "plan a little, design a little, code a little" approach has been proposed a number of times in the context of evolutionary lifecycles [15], and most agile approaches also share that an up-front plan is not a good idea. In a context like this, a big-bag enactment event at the beginning of the endeavour is not feasible. Quite to the contrary, the enactment of the methodology must proceed little by little as time goes by and new method components in the methodology are "ready" to be instantiated. Recall from Section 3.2 that most method components are clabjects, and that it is their class facet which gets instantiated at the endeavour level.

The fact that enactment is done just-in-time means that the exact situation of the endeavour can be known and analysed prior to the instantiation of a method component. For example, imagine a "Write code" task kind specified in a methodology. Using a traditional approach, this task kind would have to be instantiated at the beginning of the endeavour and assigned a start time and duration well ahead of the actual time of performance. Since exact dates and durations cannot be known beforehand with precision, rough estimates would be probably used. This is, incidentally, the same problem that conventional project plans usually find. Using a just-in-time enactment approach, the "Write code" task kind would not be instantiated until a developer actually needs to write some code; at that precise moment, an instance of the task kind is created, the situation of the whole endeavour is evaluated, and the most favourable start time and duration values are assigned to the task. Of course, anything is possible between these two extremes. Our point here is that just-in-time enactment presents an interesting alternative to the more traditional, one-off variant.

4.2 The Work Product Pool Approach

The *work product pool* can be conceived as a central repository where all the intermediary work products managed by the endeavour are stored. We must take into account that the work product pool is an abstract construct and very rarely it is actually implemented as a real database or information store. In any case, more often than not it would contain pointers to the actual work products rather than the work products themselves. Initially, when the endeavour starts, the work product pool

contains very few products, such as those that are internally available (e.g. a reusable asset database) or those that are externally provided (e.g. a needs statement). As tasks being to be carried out, and actions to be performed, existing work products are read, modified and deleted, and new work products are created, changing the population of the pool. Eventually, the final system (i.e. the ultimate goal of the endeavour) appears in the pool and the endeavour can be considered complete.

This approach sees work products as the drivers of the endeavour, and tasks as secondary elements that operate on them and transform them as necessary. This fits nicely with our discussion of process- vs. product-centric methodologies in Section 3.3. It is also highly compatible with the notion of microprocesses described by [8], and with the fountain model of [10].

A question still remains, namely, determining which method components are "ready" to be enacted at any point in time. In our previous example, we assumed that a developer needed to write some code, and that the situation of the endeavour was such that writing code was feasible. This means that the developer had the appropriate role and that all the work products that are necessary in order to write code are available in the work product pool. It is possible to build a tool that implements an algorithm that automatically finds which task kinds of a given methodology are *candidate* task kinds, i.e. are ready to be enacted if a particular user wishes. A candidate task kind is one that can be enacted at a given point in time because:

- The user requesting the enactment has a role that is mapped, as per the methodology specification, to that kind of task.
- The organisation carrying out the methodology is performing at a capability level equal or higher to that of the task kind.
- All the work products that are necessary to perform a task of the given kind are present in the pool.

An algorithm capable of verifying these three conditions is easily implementable on top of ISO/IEC 24744 because of the following characteristics of the metamodel:

- ISO/IEC 24744 contains classes that represent both the endeavour as well as the methodology domains. This allows ISO/IEC 24744-based tools to manipulate information belonging to both domains in an integrated fashion, and trace back and forth as necessary. A metamodel that only models the methodology domain will find it very difficult (if not impossible) to implement this algorithm.
- ISO/IEC 24744 directly supports the concept of capability or maturity levels as defined by standards such as CMMI [21] or ISO/IEC 15504 [11]. This provides an additional dimension to methodologies, which can be adjusted according to the desired capability of the performing organisation.
- ISO/IEC 24744 implements rich semantics for the interface between process and product aspects of the methodology, allowing the algorithm to "reason" about the dependencies between work products and therefore determine whether the required products for any particular task are present in the pool or not. A metamodel based on a more conventional input/output approach would be unable to attain this.

A tool that implements an algorithm like this would allow a software developer to display a list of candidate task kinds at any point in time and choose, from that list, any one that she wishes to enact. As we have said, which task kinds are candidate to be enacted is determined just in time depending on the role of the developer, the organisation's performing capability level, and the contents of the work product pool at that precise point in time. The major advantages of this approach as opposed to a conventional, plan-driven one, are two. First of all, it takes advantage of as much information as possible, since it defers enactment to the last possible moment, resulting in optimal decisions and a decreased need for corrections. Secondly, it is highly opportunistic, meaning that the state of the endeavour (primarily given by the contents of the work product pool) is what determines, at any point in time, the next steps that must be taken. This, in turn, supports the appearance of complex emergent behaviours, which have been described as a key component of modern business environments [14].

5 Conclusions

In this paper we have introduced the ISO/IEC 24744 standard metamodel, focussing on how it can help the implementation of a method engineering solution from both infrastructural and superstructural perspectives. On the one hand, ISO/IEC 24744 can provide the theoretical background for method engineering to model the methodology and endeavour domains together and maintain the relationships between them, which is fundamental to keep the necessary traceability during enactment. Also, this metamodel provides rich semantics to model the interface between the process and product sides of methodologies, allowing a better connection between these two often separated worlds. On the other hand, we have shown how these properties make it possible that an algorithm can be implemented on top of ISO/IEC 24744 that can determine what method components of a methodology can be enacted at any point in time. Such just-in-time enactment has the advantages over conventional, up-front enactment that it uses as much information as possibly available (requiring less rework) and that it is highly opportunistic, supporting the emergent behaviours that usually occur in modern business environments.

We acknowledge that the advantages provided by ISO/IEC 24744 come to a price: the theoretical underpinnings of the metamodel, based on powertype patterns and clabjects, depart from the conventional strict metamodelling paradigm often found in the literature. As with any other new technology, a moderately steep learning curve is to be expected. We hope that the gains will be worth it.

References

1. Atkinson, C. and T. Kühne, 2000. Meta-Level Independent Modelling. In *International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming*. 12-16 June 2000.

2. Atkinson, C. and T. Kühne, 2001. Processes and Products in a Multi-level Metamodeling Architecture. *Int. J. Software Eng. and Knowledge Eng.* **11**(6): 761-783.
3. Brinkkemper, S., 1996. Method Engineering: Engineering of Information Systems Development Methods and Tools. *Information and Software Technology*. **38**(4): 275-280.
4. Chau, T., F. Maurer, and G. Melnik, 2003. Knowledge Sharing: Agile Methods vs. Tayloristic Methods. In *12th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003)*. IEEE Computer Society. 302-307.
5. Gonzalez-Perez, C. and B. Henderson-Sellers, 2005. A Representation-Theoretical Analysis of the OMG Modelling Suite. In *The 4th International Conference on Software Methodologies, Tools and Techniques*. 28-30 September 2005. Frontiers in Artificial Intelligence and Applications 129. IOS Press: Amsterdam. 252-262.
6. Gonzalez-Perez, C. and B. Henderson-Sellers, 2006. A Powertype-Based Metamodelling Framework. *Software and Systems Modelling*. **5**(1): 72-90.
7. Gonzalez-Perez, C. and B. Henderson-Sellers, 2007. Modelling Software Development Methodologies: A Conceptual Foundation. *Journal of Systems and Software*. **(in press)**.
8. Greenfield, J. and K. Short, 2004. *Software Factories*: John Wiley & Sons.
9. Guizzardi, G., 2007. On Some Modal Properties of Ontologically Well-Founded Structural Conceptual Models. In *CAiSE 2007*. LNCS (in press). Springer-Verlag
10. Henderson-Sellers, B., 1992. *A Book of Object-Oriented Knowledge*. New York: Prentice-Hall.
11. International Organization for Standardization / International Electrotechnical Commission, 2004. ISO/IEC 15504-1: 2004. *Software Process Assessment - Part 1: Concepts and Vocabulary*.
12. International Organization for Standardization / International Electrotechnical Commission, 2007. ISO/IEC 24744. *Software Engineering - Metamodel for Development Methodologies*.
13. Kumar, K. and R.J. Welke, 1992. Methodology Engineering: a Proposal for Situation-Specific Methodology Construction, in *Challenges and Strategies for Research in Systems Development*, W.W. Cotterman and J.A. Senn (eds.). John Wiley & Sons: Chichester (UK). 257-269.
14. Lycett, M., R.D. Macredie, C. Patel, and R.J. Paul, 2003. Migrating Agile Methods to Standardized Development Practice. *IEEE Computer*. **36**(6): 79-85.
15. McConnell, S., 1996. *Rapid Development*. Redmond: Microsoft Press.
16. Object Management Group, 2005. formal/05-01-06. *Software Process Engineering Metamodel Specification*, version 1.1.
17. Object Management Group, 2005. formal/05-07-04. *Unified Modelling Language Specification: Superstructure*, version 2.
18. Object Management Group, 2006. ad/2006-08-01. *Software & Systems Process Engineering Meta-Model*, version 2.0.
19. Object Management Group, 2006. formal/05-07-05. *Unified Modelling Language Specification: Infrastructure*, version 2.
20. Seidewitz, E., 2003. What Models Mean. *IEEE Software*. **20**(5): 26-31.
21. Carnegie Mellon Software Engineering Institute, 2002. CMMI-SE/SW/IPPD/SS, V1.1, Continuous. *CMMI for Systems Engineering/Software Engineering/Integrated Product and Process Development/Supplier Sourcing, Continuous Representation*, version 1.1.
22. Thomsett, R., 2002. *Radical Project Management*. Upper Saddle River, NJ: Prentice-Hall.