Towards an Agile Foundation for the Creation and Enactment of Software Engineering Methods: The SEMAT Approach

Brian Elvesæter[1], Michael Striewe[2], Ashley McNeile[3] and Arne-Jørgen Berre[1]

[1] SINTEF ICT, P. O. Box 124 Blindern, N-0314 Oslo, Norway
{brian.elvesater, arne.j.berre}@sintef.no
[2] University of Duisburg-Essen, Gerlingstrasse 16, D-45127 Essen, Germany
michael.striewe@s3.uni-due.de
[3] Metamaxim, 48 Brunswick Gardens, London W8 4AN, UK
ashley.mcneile@metamaxim.com

Abstract. The Software Engineering Method and Theory (SEMAT) initiative seeks to develop a rigorous, theoretically sound basis for software engineering methods. In contrast to previous software engineering method frameworks that rely on separate method engineers, the primary target of SEMAT are practitioners. The goal is to give software development teams the opportunity to themselves define, refine and customize the methods and processes they use in software development. To achieve this goal SEMAT proposes a new practitioner-oriented language for software engineering methods that is focused, small, extensible and provides formally defined provides formally defined behaviour to support the conduct of a software engineering endeavour. This paper presents and discusses how the proposed language supports an agile creation and enactment of software engineering methods. The SEMAT approach is illustrated by modelling parts of the Scrum project management practice.

Keywords: Method engineering, software engineering methods, SEMAT, SPEM, ISO/IEC 24744, Scrum

1 Introduction

The Software Engineering Method and Theory (SEMAT)[1] initiative seeks to develop a rigorous, theoretically sound basis for software engineering methods. Previous software engineering frameworks and standards such as the Software and Systems Process Engineering Metamodel (SPEM) 2.0 [1] mainly target *method engineers* by providing rich but consequently often complex languages for detailed process definition; but generally not supporting enactment [2]. In contrast, the primary objective of SEMAT is to target *practitioners* (i.e., architects, designers, developers, programmers, testers, analysts, project managers, etc.) by providing a focused, small, extensi-

[1] http://www.semat.org

ble domain-specific language that allows them to create and enact software engineering methods in an agile manner.

An agile approach to software engineering methods is one that supports practitioners in dynamically adapting and customizing their methods during the preparation and execution of a project, controlled through company-specific governance, use of examples and other means. This enables practitioners to accurately document how they work and effectively share their experiences with other teams. To go beyond the current state of the practice a robust and extensible foundation for the agile creation and enactment of software engineering methods is required.

This paper presents and discusses the main new ideas and language concepts from the Essence specification [3] which has been submitted by SEMAT as a response to the Request for Proposal (RFP) "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" [4] issued by the Object Management Group (OMG). The remainder of the paper is structured as follows: In Section 2 we present and summarise the language requirements stated in the OMG RFP. In Section 3 we present the language architecture of the Essence specification and its main language constructs. In Section 4 we illustrate the SEMAT approach by modelling parts of the Scrum project management practice. Section 5 discusses this approach to related work. Finally, Section 6 concludes this paper and describes some future work.

2 Language Requirements and the Meta-Object Facility (MOF)

The OMG RFP "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" [4] solicits proposals for a foundation for the agile creation and enactment of software engineering methods. This foundation is to consist of a *kernel* of software engineering domain concepts and relationships that is extensible (scalable), flexible and easy to use, and a domain-specific modelling *language* that allows software practitioners to describe the essentials of their current and future practices and methods. In this paper we focus on the language. The requirements for the language are summarised in Table 1 below.

Table 1. Language definition (1.x) and language features (2.x) requirements

ID	Name	Description
1.1	MOF metamodel	Abstract syntax defined in MOF.
1.2	Static and operational semantics	Static and operational semantics defined in terms of the abstract syntax.
1.3	Graphical syntax	Graphical syntax that maps to the abstract syntax.
1.4	Textual syntax	Textual syntax that maps to the abstract syntax.
1.5	SPEM 2.0 metamodel reuse	Reuse SPEM 2.0 metamodel where appropriate.
2.1	Ease of use	Easy to use by practitioners.
2.2	Separation of views	Separation of two different views of a method for practitioners and method engineers.
2.3	Specification of kernel elements	Description, relationships, states, instantiation and metrics.
2.4	Specification of practices	Cross-cutting concern, element instantiation, work products, work progress, verification.

ID	Name	Description
2.5	Composition of practices	Overall concerns, merging elements, separating elements, practice substitution.
2.6	Enactment of methods	Tailoring, communication, managing, coordinating, monitoring, tooling.

The language definition (1.x) requirements mandate that the language must be compliant with the MOF metamodel architecture. The Meta-Object Facility (MOF) specification [5] serves as a foundation for the OMG Model-Driven Architecture (MDA) [6] approach and provides us with a formalism to define and integrate modelling languages. The left side of Fig. 1 illustrates the MOF four-layer architecture. MOF defines a meta-metamodel or meta-language at the M3 layer which can be used to define a metamodel or language to support method engineering at the M2 layer. A method engineering modelling language typically defines language constructs to support the definition and composition of methods out of reusable model fragments or templates. These model templates at the M1 level are typically defined by method engineers and are instantiated during a software endeavour, i.e., software development project, and used by the software practitioners (M0 level).

The right side of Fig. 1 illustrates an instance tree. MDA positions MOF as the single meta-language (M3), so there is only one top node for which different languages (M2) can be defined. Each of these modelling languages can be used to define various models (M1) of which different instantiations can be made (M0). In this paper we focus our discussion on a single domain-specific language to support method engineering. Thus in the remainder of this paper we will focus on the M2 layers and below as illustrated by the dashed boxes.

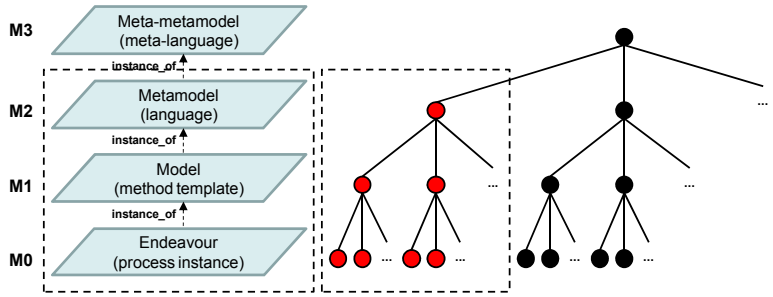

Fig. 1. MOF metamodel architecture and instantiation tree

The language features (2.x) requirements focus in particular on the ability to specify practices, compose practices into methods and enact those methods. The requirements also state that existing foundations such as the SPEM 2.0 [1] should be reused where appropriate.

3 The SEMAT Approach

The Essence specification [3], which is the initial response to the OMG RFP from the SEMAT community, defines a kernel of essentials for software engineering and an

associated language for describing practices and methods using the kernel as a standardised baseline. This approach draws inspiration from the ideas presented by Ivar Jacboson et al. in [7] which is supported by the tool EssWork[2]. The SEMAT community is iterating and improving on these ideas with the goal of defining a widely-accepted kernel and language that can be standardised by the OMG.

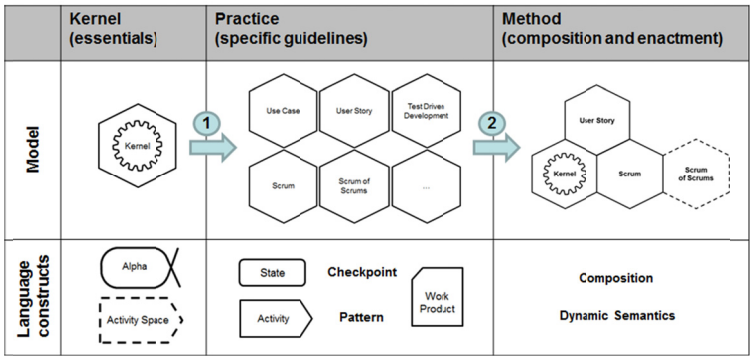

Fig. 2. Overview of the SEMAT approach

Fig. 2 illustrates the SEMAT approach and some of the key language constructs and their proposed graphical notation. A key idea is that in all software endeavours, there is a kernel, i.e., a common ground, which includes a few essential elements of software engineering that can be used as a standardised baseline for describing software engineering methods. The kernel defines a compact set of *Alphas* that represents essential things to work with (e.g., *Requirements, Work, Team, Software System,* etc.) that are universal to software engineering and a small set of *Activity Spaces* that represents the essential things to do (e.g., *Prepare to do the Work, Coordinate the Work, Support the Team,* etc.).

A Practice (e.g., *User Story, Scrum,* etc.) use the kernel as a baseline and provides specific guidelines expressed using language constructs such as *State*, *Activity*, *Checkpoint*, *Pattern* and *Work Product* addressing a particular aspect of the work at hand. The practices are composed to form a method. A definition of a method is given in [8] as "a systematic way of doing things in a particular discipline".

The language contains four parts, 1) abstract syntax, 2) composition algebra, 3) dynamic semantics and 4) graphical syntax. This paper focuses on the static and dynamic semantics which will be elaborated in the following subsections.

3.1 Static Semantics – Creation of Software Engineering Methods

The static semantics of the language specifies a metamodel that contains constructs to describe the kernel, practices and methods. The language has been structured as layers to allow for easier understanding and usage of different subsets. Fig. 3 below shows,

[2] http://www.ivarjacobson.com/process_improvement_technology/esswork

in a slightly simplified form[3], the layers of the language and the key constructs of each layer.

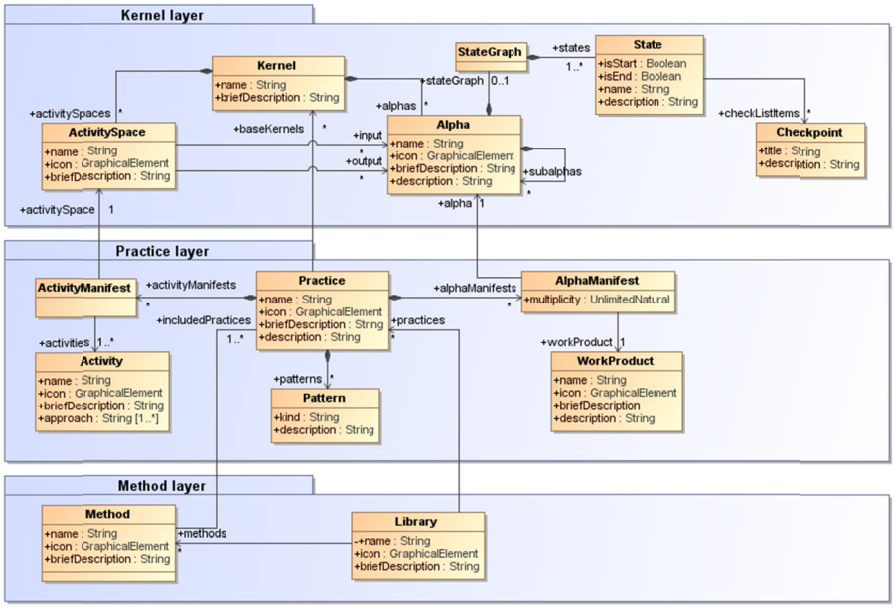

Fig. 3. Language specification – static semantics (modified and simplified metamodel excerpt)

The *Kernel layer* contains the constructs *Kernel*, *Alpha* and *Activity Space*. A *Kernel* is a set of elements used to form a common ground for describing a software engineering endeavour. *Alpha* is short for *Abstract-Level Progress Health Attribute* and represents an essential element that is relevant to the assessment of the progress and health of a software engineering endeavour. Alphas are used to describe subjects whose evolution we want to understand, monitor, direct and control. Each Alpha has a state graph with well-defined states. The states define a controlled evolution throughout its lifecycle. Each state has a collection of checkpoints that describes the criteria that the Alpha must fulfil to be in that particular state. An Alpha may also have subalphas, e.g., for splitting *Software System* into *Components* or *Requirements* into *Requirement Items*. An *Activity Space* provides a placeholder for something to be done in the software engineering endeavour. Activity Spaces frame the activities of a software engineering endeavour at an abstract level, capturing what needs to be done without defining or constraining how it is done. Activity Spaces take Alphas as input to the work. When the work is concluded the Alphas are updated and hence their states may have changed.

The *Practice layer* contains the constructs *Practice*, *Activity*, *Activity Manifest*, *Work Product*, *Alpha Manifest* and *Pattern*. A *Practice* is a general, repeatable approach to doing something with a specific purpose in mind, providing a systematic and teachable way of addressing a particular aspect of the work at hand. An *Activity*

[3] In the Essence specification, the language is actually structured as four layers, as the Practice layer is divided into two: for *simple* and *advanced* practices.

defines one or more kinds of work and gives guidance on how to perform these. An *Activity Manifest* binds a collection of activities to an activity space. A *Work Product* is an artefact of value and relevance for a software engineering endeavour. An *Alpha Manifest* binds a collection of work products to an alpha. A *Pattern* is a general mechanism for defining a structure in a practice. The practice may also specify additional alphas or extend existing alphas with sub-alphas.

The Method layer contains the constructs *Method* and *Library*. A *Method* describes how an endeavour is run. A *Library* includes a collection of practices and methods.

3.2 Dynamic Semantics – Enactment of Software Engineering Methods

In contrast to the static semantics of the language, the dynamic semantics do not specify a metamodel at the M2 layer in the MOF architecture (see Fig. 1), but rather a model at the M1 layer. This model defines abstract superclasses that contain properties for the "run-time" occurrences during the endeavour, i.e., at the M0 layer according to the MOF architecture.

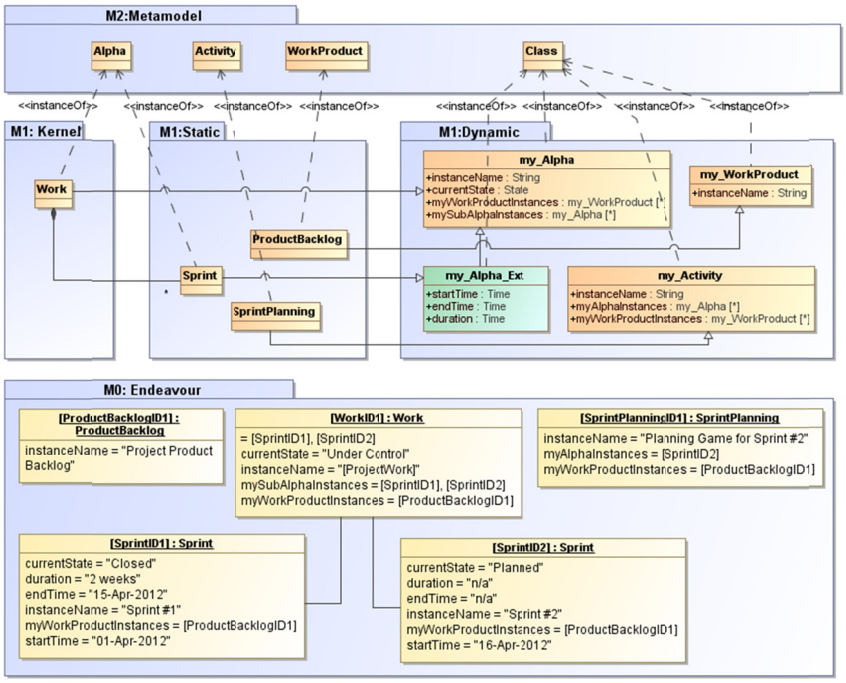

Fig. 4. Language specification – dynamic semantics

Fig. 4 illustrates the dualism of the static semantics and the dynamic semantics domain. The *M2:Metamodel*, *M1:Kernel* and a set of *M1:Dynamic* classes are part of the standard proposed by the Essence specification, while the *M1:Static* and the highlighted *_Ext* class in *M1:Dynamic* are extensions added by a practitioner. The static semantic domain defines constructs such as *Alpha*, *Activity* and *Work Product* at M2 and class instances of these metaclasses at M1 such as *Sprint*, *Sprint Planning* and

Product Backlog. The dynamic semantic domain defines superclasses such as *my_Alpha*, *my_Activity* and *my_WorkProduct* for the respective *Alpha*, *Activity* and *Work Product* class instances from the static semantic domain.

The generalization mechanism ensures that the instances on the endeavour M0 layer contain slots that have been defined both from the static semantic and the dynamic semantic domains. Using this mechanism one could also define specific extensions, such as run-time endeavour properties that should only apply to certain subclasses of alphas, e.g., *my_Alpha_Ext* that contains properties such as *stateTime*, *endTime* and *duration* that should apply to *Sprint* instances that are sub-alphas of *Work* that is defined in the kernel. Moreover, dynamic semantics can be formalized with this mechanism by defining operations using the superclasses of the dynamic semantic domain.

The relationship between the static elements and their dynamic counterparts can be supported by tools and services that allow precise definition of these associations. Having such mechanisms in place will provide practitioners the ability to use the methods as described, but also refine and customize the methods according to the needs of the endeavour.

4 Illustrative Approach – Scrum

This section illustrates the SEMAT approach by modelling selected parts of the Scrum project management practice [9] and explores how the Scrum practice may be mapped to the Essence Kernel and Language. A particular *Method* can be composed from a set of well-defined *Practices* that plugs into the *Kernel*. Composition and usage of the method is supported by language's composition and dynamic semantics. The Scrum practice defines specific work products and activities that can be associated with the alphas and activity spaces defined by the kernel.

4.1 Creating the Scrum Practice and Method

In this paper we only present a subset of the Essence Kernel in order to illustrate the principles. Fig. 5 represents a subset of the alphas and activity spaces defined in the Essence specification [3]. The Kernel defines a small set of universal alpha elements such as *Requirements*, *Software System*, *Work* and *Team,* and their relationships, and activity spaces for the endeavour such as *Prepare to do the Work*, *Coordinate the Work*, *Support the Team*, *Track Progress* and *Stop the Work*.

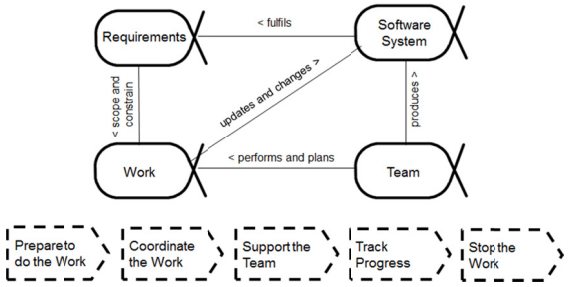

Fig. 5. Alpha and Activity Space subset defined by the Kernel

Since the paper focuses on the enactment part we only elaborate the details of the *Work* alpha. Fig. 6 shows the states of work, the corresponding state descriptions and the checkpoints associated with each state.

States (State Graph)	Description (State)	Checklist items (Checkpoints)
Initiated	The work has been requested.	• The result required of the work being initiated is clear. • Any constraints on the work's performance are clearly identified. • …
Prepared	All pre-conditions for starting the work have been met.	• Commitment is made. • Cost and effort of the work are estimated. • …
Started	The work is proceeding.	• Development work has been started • Work progress is monitored. • …
Under Control	The work is going well, risks are under control, and productivity levels are sufficient to achieve a satisfactory result.	• Work items are being completed. • Unplanned work is under control. • …
Concluded	The work to produce the results has been concluded.	• All outstanding work items are administrative housekeeping or related to preparing the next piece of work. • Work results are being achieved. • …
Closed	All remaining housekeeping tasks have been completed and the work has been officially closed.	• Lessons learned have been itemized, recorded and discussed. • Metrics have been made available. • …

Fig. 6. States of work, state descriptions and associated checkpoints

A checkpoint describes the entry criteria for entering a specific state of the alpha and can be used to measure the progress of the alpha in the software endeavour. Checkpoints are typically expressed in natural language and provide a basis for generation of descriptive checklists items that are interpreted by the practitioners.

We extend the Work alpha for Scrum (see left side of Fig. 7). The Work alpha is typically used for the duration of a development project that may cover a number of sprints. Thus we define a new sub-alpha called Sprint. The Sprint Backlog is modelled as a work product that is associated with the Sprint sub-alpha. The Sprint has its own state graph (see middle part of Fig. 7). Scrum comes with its own specific set of rules that should be defined as part of the practice, whereas the Work state machine and its associated checkpoints are more general. The identified Scrum events may be mapped to corresponding activities and associated with the proper activity spaces (see right side of Fig. 7).

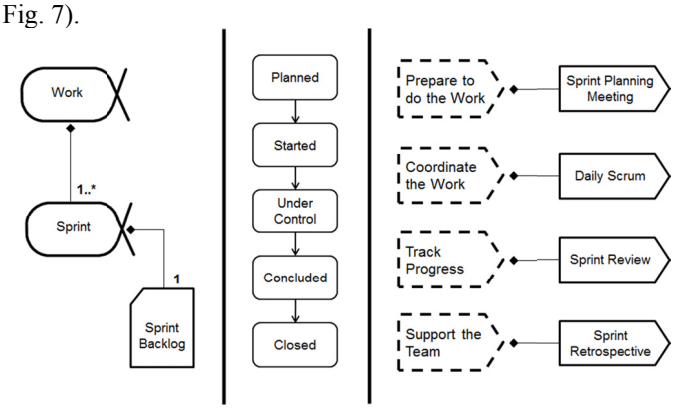

Fig. 7. The Sprint sub-alpha extends the Work alpha (left), the states of the Sprint sub-alpha (middle) and mapping of Scrum activities to the Activity Spaces of the Kernel (right)

4.2 Enacting the Scrum Method

The enactment support in the Essence language recognizes that software development is a creative process and most of the progress within the software development endeavour is done by human agents [10]. Thus, process enactment of software engineering methods and practices must acknowledge the human knowledge worker and provide means of monitoring and progressing the software endeavour through human agents foremost and automation secondly. In enactment, a team of practitioners collaborates on decision-making, planning and execution. Enactment may be partially supported by method repositories and process engines that are linked to tools such as project management and issue tracking systems.

The concrete syntax of the language has been designed so that the usage of the composed practices, i.e., the method, should be easy for the practitioners. For this purpose, the concept of *Alpha state cards* is introduced. Fig. 8 below shows the state card for the Sprint alpha at the *initial* state (left) and in the *Planned* state (right). These cards can be used for reading and understanding the practice, and also how to progress the states of the Sprint according to the checklist defined. This requires that all checkpoints are ticked off.

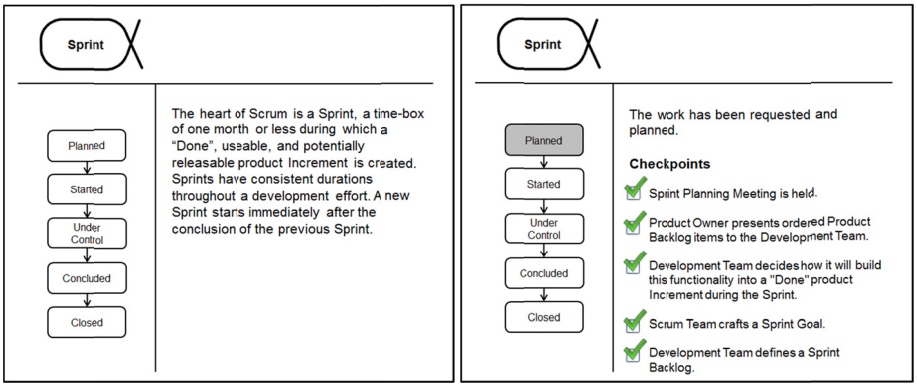

Fig. 8. Sprint state card (initial state and planned state)

Using the checkpoints of the Alphas it is at the discretion of the team to decide when a state change has occurred. These state cards may also have different views so that instead of the checklist items one could get a list of activities to do in order to progress from one state to another, or which work products to produce. The language supports the definition of different views suitable for different kinds of practitioners.

5 Related Work and Discussions

One main criticism of SPEM 2.0 is the lack of enactment support [8, 11]. Enacting SPEM processes are typically done through mapping, e.g., (1) mapping processes into project plans and enacting these with project planning and enactment systems or (2) mapping processes to a business flow or execution language and executing this with a workflow engine [1]. Designing native dynamic semantics into the language is argua-

bly one of the main requirements that will require a redesign of the SPEM architecture.

The Situational Method Engineering (SME) community [12] has been working on related metamodelling approaches which has resulted in the ISO/IEC 24744 standard [13] that provides an alternative to the OMG SPEM 2.0 specification. Fig. 9 shows the key classes in the ISO 24744 metamodel. The metamodel can be seen as a dual metamodel that defines *Methodology elements* and *Endeavour elements* that are linked together through the concept of powertypes [14]. An explanation of powertypes to model software development methods is described in [11]. There are two types of methodology element classes, namely *Template* and *Resource*. Endeavour elements which includes *Stage*, *WorkUnit*, *WorkProduct* and *Producer,* always have a corresponding methodology element *...Kind* type represented using a powertype relationship. Resource constructs, which include *Language*, *Notation*, *Constraint*, *Outcome* and *Guideline*, represents elements that are directly used without requiring the need for instantiation at the endeavour level.

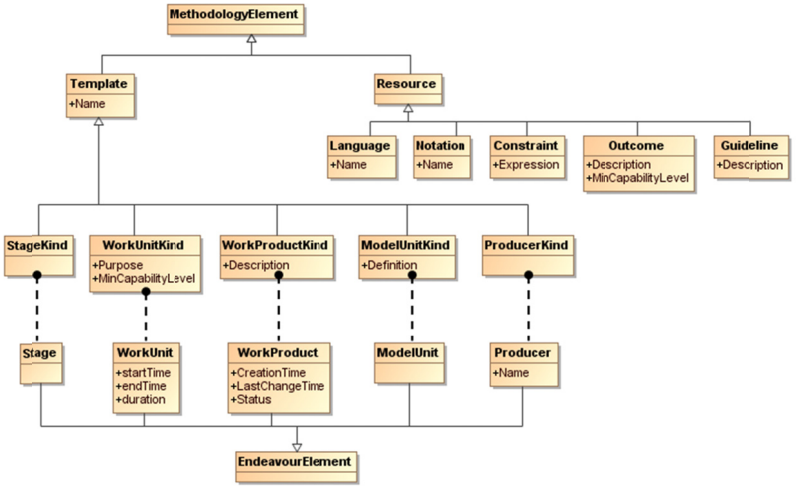

Fig. 9. Overview of the ISO 24744 metamodel

Since advanced metamodelling constructs such as powertypes are not compatible with MOF the ISO 24744 standard uses a different metamodel architecture [11, 13] to support dynamic semantics. This metamodelling issue is not present in SPEM as it leaves out the endeavour M0 layer. In the SEMAT approach we propose a solution that is MOF compliant and define two separate domains, the metamodel of the static semantics and the model of the dynamic semantics. We compose these two domains at the M1 layer of the MOF architecture (see Fig. 4) to support both method and endeavour properties at the endeavour M0 layer. This is very similar to the concept of using powertypes but maintains compatibility with the MOF architecture. Fig. 10 illustrates the difference between the two approaches. The *WorkProduct* and *WorkProductKind* metaclasses from the ISO/IEC 24744 standard are equivalent to the *WorkProduct* (at M2 layer) and *my_WorkProduct* (at M1 layer) in the Essence specification. Standardised endeavour properties on *WorkProduct* (ISO 24744) can be

represented as properties on *my_WorkProduct* (Essence). Additional user-specific extensions and properties such as *version* on ProductBacklog (ISO 24744) can be done through extensions of the domain classes as shown using *my_WorkProduct_Ext* (Essence). The resulting slots in the *ProductBacklog instance* at M0 are the same in both approaches.

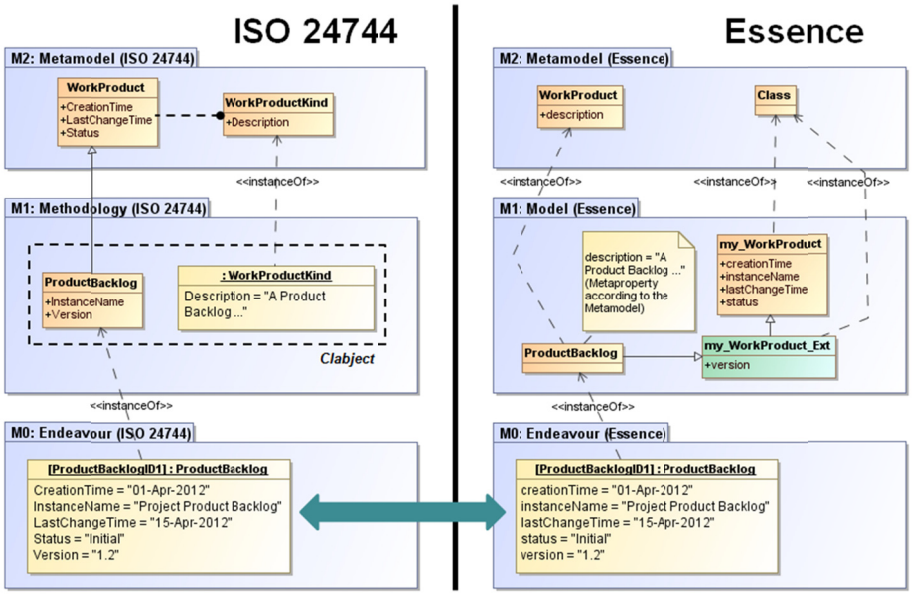

Fig. 10. Comparison of the ISO 24744 and Essence approaches

6 Conclusions and Future Work

In this paper we have presented the main ideas of the Essence language submitted by SEMAT as a response to the OMG RFP "A Foundation for the Agile Creation and Enactment of Software Engineering Methods" [4]. This paper has explained the static and dynamic semantics of the Essence language and illustrated the enactment support by modelling selected parts of the Scrum project management practice.

The proposed language contains a static semantics part defined as a metamodel on the M2 layer in the MOF architecture and a dynamic semantics part defined as a model on the M1 layer in the MOF architecture. The proposed language constructs are intended to simplify software engineering method modelling adaptation for practitioners. The concepts of *Kernel*, *Alpha*, *Activity Space* and *Practice* are introduced as examples of such required language constructs. The kernel elements form a domain model for software engineering and provide a standardised baseline for describing current and future practices that afterwards are adapted and customized for usage, i.e., enactment, in software endeavours. Our discussion has tried to clarify the differences and similarities between the Essence approach and other related standards such as the ISO/IEC 24744.

We are currently progressing this work in the context of SEMAT where we are preparing a revised submission to the OMG RFP. The aim is to take advantage of recent development and experiences from software engineering and method engineering communities in order to give the best possible direct support towards software practitioners.

Acknowledgments. This research was co-funded by the European Union in the frame of the NEFFICS project (FP7-ICT-258076) (www.neffics.eu) and the REMICS project (FP7-ICT-257793) (www.remics.eu), and SINTEF in the frame of the SiSaS project (sisas.modelbased.net). The authors acknowledge and thank collaboration with colleagues within SEMAT (www.semat.org) for foundational input and feedback.

References

[1] OMG, "Software & Systems Process Engineering Meta-Model Specification, Version 2.0", Object Management Group (OMG), Document formal/2008-04-01, April 2008. http://www.omg.org/spec/SPEM/2.0/PDF/

[2] R. Bendraou, B. Combemale, X. Cregut, and M.-P. Gervais, "Definition of an Executable SPEM 2.0", in 14th Asia-Pacific Software Engineering Conference (APSEC '07). Nagoya, Japan, 2007, pp. 390-397. http://dx.doi.org/10.1109/APSEC.2007.38

[3] OMG, "Essence - Kernel and Language for Software Engineering Methods, Initial Submission - Version 1.0", Object Management Group (OMG), OMG Document ad/12-02-04, 20 February 2012. http://www.omg.org/cgi-bin/doc?ad/12-02-04

[4] OMG, "A Foundation for the Agile Creation and Enactment of Software Engineering Methods RFP", Object Management Group (OMG), OMG Document ad/2011-06-26, 23 June 2011. http://www.omg.org/members/cgi-bin/doc?ad/11-06-26.pdf

[5] OMG, "Meta Object Facility (MOF) Core Specification, Version 2.0", Object Management Group (OMG), Document formal/06-01-01, January 2006. http://www.omg.org/spec/MOF/2.0/PDF/

[6] OMG, "MDA Guide Version 1.0.1", Object Management Group (OMG), Document omg/03-06-01, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf

[7] I. Jacobson, P. W. Ng, and I. Spence, "Enough of Processes - Lets do Practices", Journal of Object Technology, vol. 6, no. 6, pp. 41-66, 2007. http://www.jot.fm/issues/issue_2007_07/column5

[8] C. Gonzalez-Perez and B. Henderson-Sellers, "Metamodelling for Software Engineering", John Wiley & Sons, Ltd, 2008, ISBN 978-0-470-03036-3.

[9] K. Schwaber and J. Sutherland, "The Scrum Guide", Scrum.org, October 2011. http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf

[10] P. Feiler and W. Humphrey, "Software Process Development and Enactment: Concepts and Definitions", Software Engineering Institute, Technical report CMU/SEI-92-TR-004, September 1992.

[11] B. Henderson-Sellers and C. Gonzalez-Perez, "The Rationale of Powertype-based Metamodelling to Underpin Software Development Methodologies", in Proc. of the 2nd Asia-Pacific conference on Conceptual modelling - Volume 43, 2005, ACM.

[12] M. A. Jeusfeld, M. Jarke, and J. Mylopoulos, "Metamodeling for Method Engineering", The MIT Press, 2009.

[13] ISO/IEC, "Software Engineering – Metamodel for Development Methodologies", International Organization for Standardisation (ISO), ISO/IEC 24744, 15 February 2007.

[14] J. Odell, "Power Types", Journal of Object-Oriented Programming, vol. 7, no. 2, pp. 8-12, 1994.