Process Patterns for MDA-Based Software Development

Mohsen Asadi, Naeem Esfahani, Raman Ramsin
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran
E-mail: mohsenasadi@mehr.sharif.edu, esfahani@ce.sharif.edu, ramsin@sharif.edu

Abstract—**Information systems are expected to satisfy increasingly ambitious requirements, while reducing time–to–market has become a primary objective. This trend has necessitated the advent of development approaches that are better equipped and flexible enough to cope with modern challenges. Model-Driven Architecture (MDA) and Situational Method Engineering (SME) are approaches addressing this requirement: MDA provides promising means for automating the software process, and revitalizes the role of modeling in software development; SME focuses on project-specific methodology construction, mainly through assembling reusable method fragments (process patterns) retrieved from a method base. We provide a set of high-level process patterns for model-driven development which have been derived from a study of six prominent MDA-based methodologies, and which form the basis for a proposed generic MDA Software Process (MDASP). These process patterns can promote SME by providing classes of common process components which can be used for assembling, tailoring, and extending MDA-based methodologies.**

Keywords-Situational Method Engineering, Model-Driven Development, Process Patterns

I. INTRODUCTION

Most software systems cannot be built from scratch anymore, and as systems get ever more complex, reuse is becoming increasingly important. Promoting reusability is almost impossible without abstraction; irrelevant details should be shaved off if reusability is to be achieved. There are several trends that are trying to make software engineering concepts and artifacts more abstract, and we can clearly see the emergence of research paradigms around these trends.

As a software engineering approach, Model-Driven Development (MDD) is one of the results of these trends. The main goals in MDD are portability, interoperability, and reusability; in order to achieve these goals, developers create and evolve the software at different levels of abstraction – each corresponding to a layer – while transitions between layers are meant to be automatic. The Model-Driven Architecture (MDA) is a particular realization of MDD, at the core of which several standards and modeling approaches have been introduced to enable users to create, manage and translate the models produced [1]. However, MDA incorporates no concrete process for software development. Hence, developers should either adapt existing traditional methodologies to make them suitable for use in an MDA context, or use new methodologies especially devised to support MDA rules and standards. In MDA-based development, the main model produced is a Platform Independent Model (PIM), later transformed into one or more Platform Specific Models (PSMs). The major benefit of this approach is that the PIM is isolated from the platform; abstractions are thus separated from implementations.

Another outcome of abstraction trends in software engineering is the pattern-based approach to software systems modeling and development [2]. Architectural- and design patterns are particularly well-known in this context; however, patterns have also been defined and used in the context of software processes themselves, giving rise to process patterns [2, 3]. Situational Method Engineering (SME) approaches, which focus on project-specific construction of methodologies, have much benefited from the notion of process patterns, as SME is mainly applied through assembling reusable method fragments (akin to process patterns) which are retrieved from a method base. SME thus provides a degree of flexibility which is far superior to that supported by heavyweight processes such as the Rational Unified Process (RUP) [4].

Since the idea behind both MDD and the Pattern movement is the promotion of abstraction and reuse, extracting process patterns from MDA-based methodologies will help achieve an even greater level of abstraction. We provide a set of high-level process patterns for MDA-based model-driven development, and a generic process model for MDA-based methodologies. The patterns have been derived from a study of six prominent MDA-based methodologies, and can promote situational method engineering by providing classes of common process components.

This paper is organized as follows: The general framework for pattern-based processes is described in the next section; Section 3 defines the MDA Software Process (MDASP) as an instance of the proposed framework, specifically targeting the MDA context; Section 4 introduces the process patterns derived from the MDASP and six major MDA-based methodologies;

Section 5 shows how the MDASP corresponds to the six methodologies used as pattern sources; Section 6 discusses the potential applications of the proposed MDA-based process patterns, and Section 7 contains the conclusions and some suggestions for future work.

II. A Pattern-based Process Framework

Process patterns should adhere to a standard if they are to be retrieved and reused effectively; such a standard, however, does not exist. Some methodologies and method composition/configuration approaches [5] have incorporated the concept of process patterns, and have defined a template for defining and applying them. Gnatz *et al.* have focused on defining a process framework for the definition and use of process patterns [6]. We have refined and restructured the framework proposed in [6] through applying the layered architecture proposed in [2]. The resulting pattern-based process framework (Fig. 1) has been used for instantiating our generic MDA-based software process.

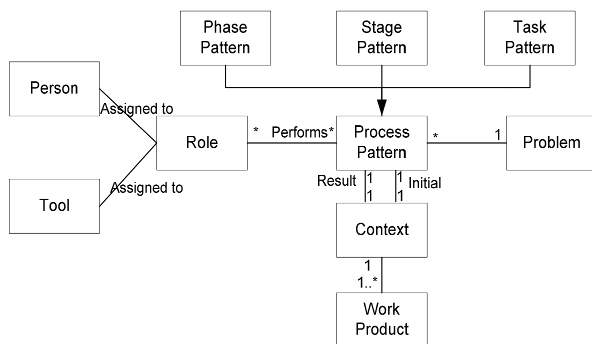

Figure 1. Pattern-based process framework

The patterns are organized in three layers [2]. In the bottom layer, *task process patterns* reside, depicting the detailed steps required to perform a specific low-level action. A *stage process pattern* is composed of a number of task process patterns. These patterns define the steps required for completing one stage of the process. *Phase process patterns* depict the interactions occurring among constituent stage process patterns to complete a phase of the process.

Each and every pattern has a *problem* description, *roles* by which it is performed, and initial/result *contexts*. The *problem* is a high level description of why this pattern is required and what will be solved by applying it. A *role* may be assigned to a *person* or *tool* to apply a pattern. Finally, the *context* describes the configuration of artifacts before and after the execution of the pattern. In this paper, we have described all of these elements (except for roles) for each of the patterns introduced. Moreover, we have also listed the input and output artifacts.

III. MDA Software Process (MDASP)

The general framework defined above has been used in devising an MDA Software Process (MDASP). The MDASP (Fig. 2) is a generic process model for MDA-based methodologies. Its constituent process patterns have been extracted through a comprehensive study of six prominent MDA-based methodologies: MODA-TEL [7], MASTER [8], C^3 [9], ODAC [10], DREAM [11], and DRIP-Catalyst [12].

MDASP includes three serial phases, which in turn consist of internal iterative stages. The first phase initiates the project and provides the resources necessary for commencing the development. The software is then developed and deployed into the user environment through enacting the other two phases. Umbrella activities are performed throughout the entire lifecycle; we have therefore used the convention proposed in [2] and [13] for depicting them, showing them on an arrow spanning the whole lifecycle.

There is no similar generic process-pattern-based software process model for the MDA context; however, this model has been inspired by the process-pattern-based Object-Oriented Software Process (OOSP) proposed in [2], and the Agile Software Process (ASP) proposed in [13]. OOSP is more general and consequently more abstract in comparison with MDASP: OOSP spans all object-oriented methodologies regardless of their types [2], whereas in MDASP, patterns are limited to those found in MDA-based development methodologies. Therefore, MDASP differs greatly from OOSP in structure and pattern content. The differences mainly arise from the principles that define MDA-based development: For example, the automatic transformation of the PIM to the PSM, verification/validation of models, and automatic transformation of the PSM to code are most meaningful in an MDA context. On the other hand, MDASP differs from ASP [13] as to underlying concepts: MDA and all the methodologies based on it are model-driven, with the models being the key artifacts of software construction, whereas ASP and most agile methodologies are model-phobic. The following sections provide a more detailed description of our proposed MDA-based process patterns.

IV. MDASP Process Patterns

In this section, the proposed process patterns are described in detail. These patterns constitute the MDASP, and have been classified according to the pattern-based software development framework; i.e., as phase-, stage-, and task process patterns.

A. *Phase Process Patterns*

MDASP consists of three serial phases, each of which is a phase process pattern in its own right:

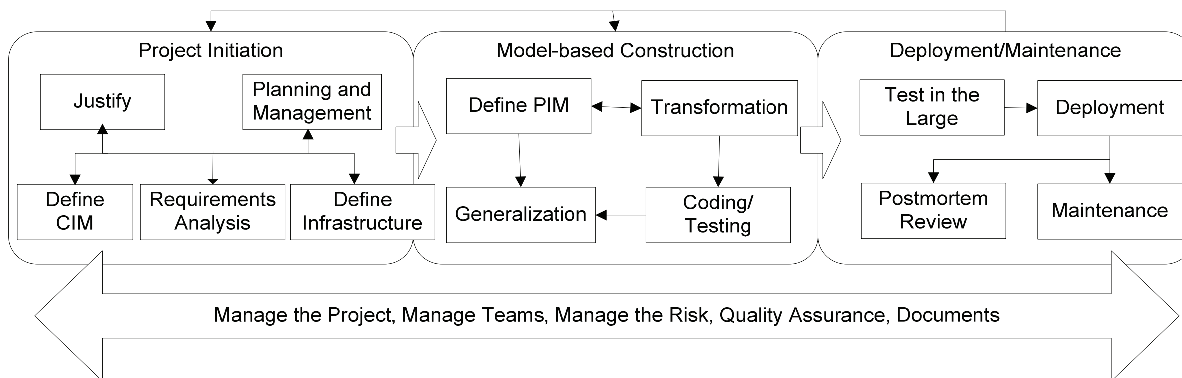

Figure 2. The proposed MDA Software Process (MDASP)

Project Initiation. The main goal of this phase is to provide a foundation for a successful software development endeavor. It provides justification for the project, and produces the requirements model, the required infrastructure, an initial plan, and management documents. The constituent stages of this phase (*CIM Definition*, *Requirements Analysis*, *Justification*, *Planning and Management*, and *Infrastructure Setup*) are performed iteratively.

Model-Based Construction. This phase produces the software in a model-driven manner. A complete and precise model of the structure and behavior of the system – the Platform Independent Model (PIM) – is first created based on the Requirements Model. The PIM is then transformed into the Platform-Specific Model (PSM), and finally into an executable release. The *Define PIM* stage creates the PIM via applying object-oriented analysis and design techniques. The *Transformation* stage checks the validity of the model through applying model checking techniques, and then refines the model according to the specific particulars of the platform. The resulting PSM is consequently transformed into code. The code thus created cannot be complete, so the *Coding and Testing* stage completes the code and performs unit testing. The *Generalization* stage abstracts and stores the products for future reuse.

Deployment and Maintenance. The objectives of this phase are to successfully deliver the developed system to the user and to keep the system in production afterwards. The developers need to perform *Testing in the Large* on the application by using system-level testing techniques. The application is then deployed into the user environment through the *Deploy* stage. After deployment, users should be supported through providing training, consultation, and system *Maintenance*. A *Postmortem* review is also conducted, through which the development process is improved, and lessons learned from the project are documented.

B. *Stage Process Patterns*

Stage process patterns comprise the bulk of phase process patterns. The objectives of phase patterns are realized through the interaction of their constituent stage patterns. Most stage patterns are executed iteratively. The stage process patterns that constitute the different phases of MDASP are explained throughout the rest of this section.

Justify. The objective is to justify the project by performing a feasibility study, and also to provide the necessary resources based on the requirements, project documents, project scope, customer viewpoints, and previous experiences (Fig. 3). Feasibility study is performed through analyzing the *Financial, Technical, Operational, Human-factor,* and *Resource-plan* feasibilities of the project, each of which corresponds to a task pattern in the *Justify* stage pattern. At the end of the stage, the *Garnering Initial Support* task obtains customer approval and support for starting the project.

Define CIM. The Computation Independent Model (CIM) is a contextual model of the problem domain, based on which the system's PIM is later produced. While the PIM considers the system as a software-intensive one, CIM is not a model of the *software* system, but an *essential* model complementing the requirements through delineating the scope of the system in the problem domain, based on customer viewpoints and project descriptions. The *Extract System Objectives*, *Describe System Scope*, *Identify High Level Services*, and *Identify External Users* tasks are the main activities of this stage (Fig. 4).

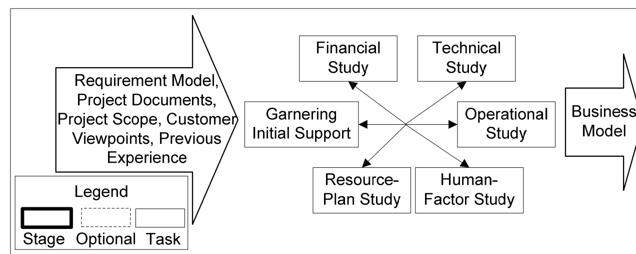

Figure 3. Components of *Justify* stage pattern

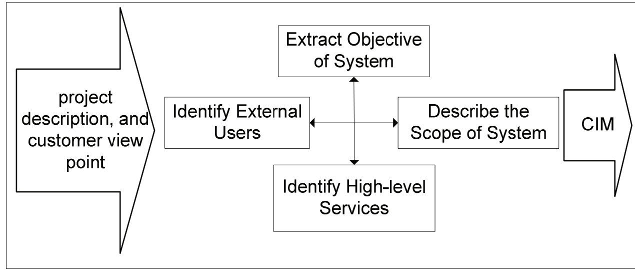

Figure 4. Components of *Define CIM* stage pattern

Requirements Analysis. The aim of this stage is to define the requirements model, in which each requirement has a unique and unambiguous definition (Fig. 5). Knowledge of existing applications, customer viewpoints, project vision and documents, and the business case are the inputs to this stage. To realize the aim of this stage, the following tasks are performed: *Capture User Requirements*, which elicits and documents the requirements; *Refine Requirements,* which aggregates, decomposes, and alternates the requirements; *Develop Requirements Model,* which uses requirements documents to define a model of the requirements, depicting the capabilities (functional requirements) and the enforcers (non functional requirements) of the system; and *Requirements Prioritization* prioritizes the requirements through applying prioritization techniques such as MoSCoW rules [14]. Requirements documents and requirements models are produced as output.

Define Infrastructure. The define-infrastructure stage provides the foundation (resources) necessary to complete the project successfully (Fig. 6). The inputs to this stage include the requirements document, business case, project description, and experiences gained from previous projects. The *Define and Organize Initial Team* task forms the development team. It is not necessary to have a complete team from the start, as the team can be reorganized during the development process. The *Tool Selection* task identifies the appropriate tools for developing the system. Since development is model-driven and some tasks are to be performed automatically, general tools (such as modeling tools, documentation tools and project management tools) and MDA-based tools (that perform model transformation and code generation) are both needed. The *Select Platform* and *Specify Transformation Type* tasks select the final platform of the system and the type of the transformation applied, respectively. Metadata is managed by the *Metadata Management* task.

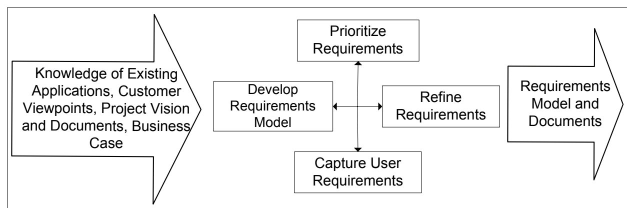

Figure 5. Components of *Requirements Analysis* stage pattern

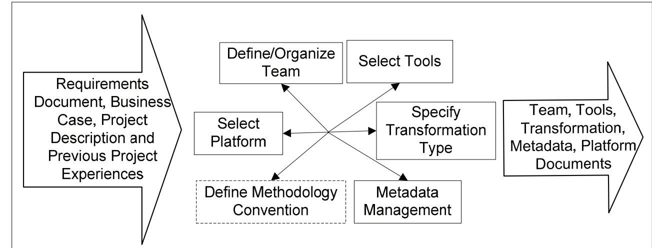

Figure 6. Components of *Define Infrastructure* stage pattern

The *Define Methodology Conventions* task is not found in any of the existing MDA-based methodologies, yet it is considered as an essential activity in some other types of methodologies (such as agile methods). It has therefore been specified as optional. The team definition, tools selection, transformation documents, metadata documents, and platform selection documents are produced as output.

Planning and Management. This stage produces the initial plan of the project as well as the initial management documents (Fig. 7). It receives the project infrastructure, initial requirements, project objectives, and feasibility-study results as input, and produces the project plan, risk assessment, and initial management documents. The *Resource and Effort Estimation* task produces the list of project tasks and the resources required. The *Time Estimation* task predicts the time needed for performing each task (assumptions and constraints are also documented). The *Define Initial Management* task documents all the information needed for project management (such as the project plan, project schedule, and communication paths). The *Risk Assessment* task identifies the risks and their priorities, and suggests strategies for mitigating them.

Define PIM. The objective of this stage is to model the detailed structure and behavior of the system without any consideration given to platform specifications (Fig. 8). Furthermore, remaining requirements are discovered and the requirements model is completed. The PIM produced is a blueprint of the software system that shows how the system functions. The requirements model, management documents, and project infrastructure are the inputs to this stage. The *Develop Analysis Model* task uses the requirements model to create the internal view of the system without any technological details, maintaining separation of concerns between functional and non-functional aspects. The *Design Architecture* task defines an architecture for the application, and specifies the relationships between the main components of the system.

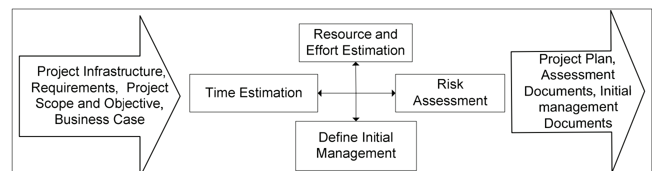

Figure 7. Components of *Planning and Management* stage pattern

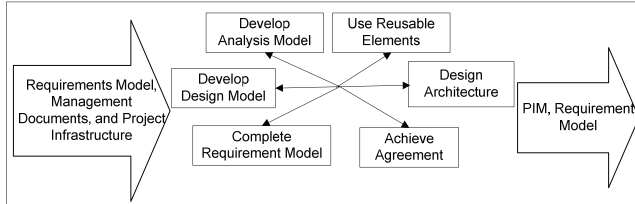

Figure 8. Components of *Define PIM* stage pattern

The *Develop Design Model* task uses the analysis model and architecture to create the design model by detailing the analysis model and adding complementary parts. For each of the models produced, repositories of reusable models (compiled from previous projects) are examined so that utmost reuse is made of models/frameworks. Team members should then *Achieve Agreement* on the PIM produced.

Transformation. One of the main objectives of the MDA is to maximize automatic generation of the deliverables. To achieve this objective, it provides certain methods for transforming abstract models to their concrete counterparts, and ultimately, to executable code. Most of the activities of this stage (Fig. 9) are performed through MDA tools. Inputs to this stage are the PIM, transformation documents, platform specifications, requirements model, and management document, and the executable system is produced as output. The *Verification/Validation* task aims at correcting PIM errors prior to transformation into the PSM. The *Rules Modification and Extension* task is where transformation rules are changed in cases where tools do not support the required mapping rules, or if a specific goal requires the definition of different rules. The *Transform PIM to PSM* task creates the PSM from the PIM by using tools. Before transformation, platform particulars and transformation rules must be set up in the tool. The *PSM to Code Transformation* task produces the executable code from the PSM. The *Check Traceability* task is performed after transformation to ensure consistency between source and target models.

Coding/Testing. The objective of this stage is to produce the complete executable code (Fig. 10). Since current MDA tools cannot generate complete code from the PSM, we need to complete the generated code manually. Unit testing is required in order to check the correctness of the code. The inputs to this stage include the generated code, requirements model, PIM, and PSM. The *Complete Code* task assigns incomplete parts of the code to the developers.

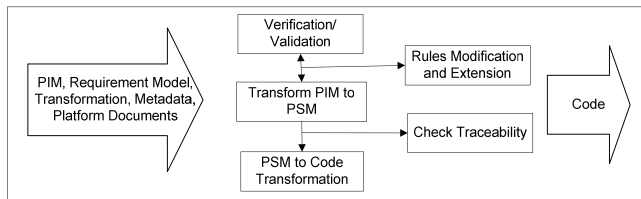

Figure 9. Components of *Transformation* stage pattern

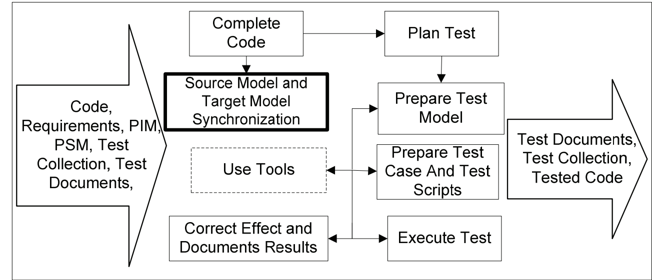

Figure 10. Components of *Coding/Testing* stage pattern

The developers then complete the code according to the PIM and PSM. The code must be synchronized with the PSM, and the PSM should be synchronized with the PIM. The *Source model and Target model Synchronization* stage propagates code changes to the PSM and PIM. We will further explain this stage in the next section. Every coded part must be tested after completion. These tests are not at the system level, and typically consist of unit tests, black box tests, regression tests, and integration tests. The tasks related to testing include: *Plan Tests, Prepare Test Model, Prepare Test Cases and Test Scripts, Execute Tests,* and *Correct Defects and Document Test Results*. Automatic testing by test tools is possible to some extent, but manual testing is usually necessary in order to complement tool-based testing.

Source Model and Target Model Synchronization. This stage aims at detecting and resolving the inconsistencies between models (Fig. 11). Inconsistencies exist between the PIM and the PSM, and also between the PSM and the code. Round-trip engineering is an approach for maintaining consistency between changing software artifacts. In order to support round-trip engineering in MDA, certain tasks are performed by the developers. This stage receives source and destination models as input. The *Define Inconsistencies* task describes consistency and inconsistency semantics in an MDA context. The *Specify Inconsistencies* task detects the inconsistencies that should be resolved. The *Define Rectify Strategy* task produces strategies for rectifying the inconsistencies. The strategies are then applied, as needed, to produce the synchronized models as output.

Test in the Large. The objective of this stage is to perform final system- and acceptance tests, and to act on the defects detected (Fig. 12).

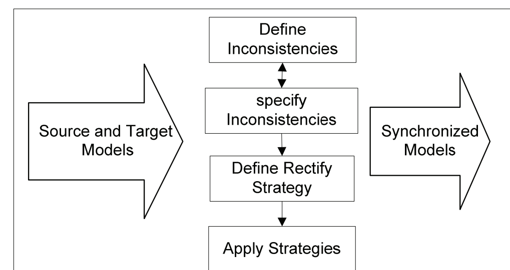

Figure 11. Components of *Source Model and Target Model Synchronization* stage pattern

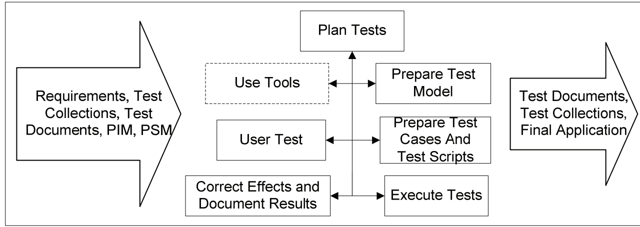

Figure 12. Components of *Test in the Large* stage pattern

Testing in the large consists of testing techniques such as function testing, system testing, user acceptance testing, stress testing, operations testing, and alpha/beta/pilot testing [2]. The requirements, test documents, PIM, PSM, and generated code are the inputs to this stage.

Generalization. The generalization stage is essential to an organization's reuse efforts, as it forces project managers to make time for making the artifacts reusable [2]. Reusability in MDA is manifest in the remapping of the PIM to different PSMs. Another essential type of reusability is the reuse of existing model components to create the PIM and PSM; this type of reusability is not directly addressed by MDA, and is deferred to methodologies instead. The generalization stage aims at making the models (including the code) reusable, so that they can be used either in the current project or in future projects. The work products created during previous stages are received as input (Fig. 13). The *Identify Reusable Work Product* task finds the potentially reusable artifacts, the *Make Work Product Reusable* task performs abstraction through holding generalization sessions, and the *Document and Store Reusable Work Product* task tags the reusable components and stores them in repositories for future reuse.

Postmortem Review. This stage receives the management document, project plan and infrastructure as input, analyzes the outcome of the project, and documents the lessons learned for use in future projects (Fig. 14). Incomplete documents created during the development process are completed by the *Project Documentation Completion* task. Initial estimates are then compared to the actual values in the current state of the project, and the methods practiced in the project are analyzed. Team members are assessed and rewarded appropriately. A training plan is outlined to address skill deficiencies in the development team(s).

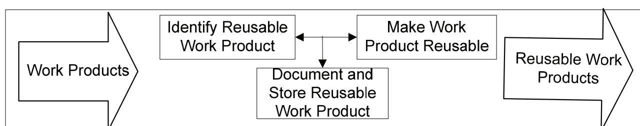

Figure 13. Components of *Generalization* stage pattern

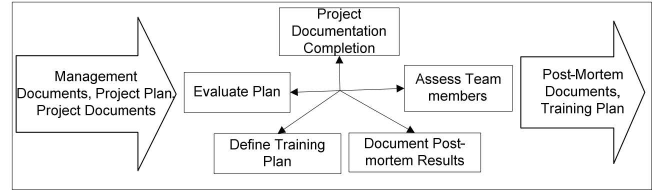

Figure 14. Components of *Postmortem Review* stage pattern

Deployment. This stage aims to deliver the developed system to the end user. It receives the final system and project documents as input, and through applying the *Prepare User Documents and Train Users, Set Up the User Environment,* and *Transition to User Environment* tasks, delivers the system to the end user (Fig. 15). This stage must be performed with strict attention to the constraints delineated in the project infrastructure.

Maintenance. The objective of this stage is to keep the system in production after deployment (Fig. 16). Users are supported through the *Support* task. Correction and enhancement is handled by the *Identify Defect and Enhancement* task. *Evaluate Functionality* checks whether the system satisfies the requirements.

V. Realization of the Proposed Process Patterns in MDA-based Methodologies

In order to verify MDASP, we have studied the mutual correspondence between the generic process and the MDA-based methodologies used as resources.

Table 1 shows how the phases of the six MDA-based methodologies correspond to (*realize*) the proposed phase- and stage process patterns. The realization table shows that the proposed process patterns do indeed cover the methodologies used as the bases.

VI. Applications of the Proposed MDA Process Patterns

Situational Method Engineering is concerned with the construction/adaptation of a methodology according to the characteristics of the project situation at hand [10]. Two well-known approaches of SME are *assembly-based* and *paradigm-based* [5]. The assembly-based approach constructs the target methodology or enhances an existing methodology through reusing process components. The paradigm-based approach instantiates, abstracts or adapts an existing meta-model to produce the target methodology.

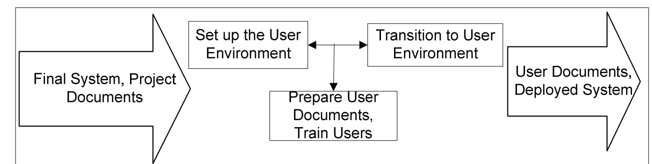

Figure 15. Components of *Deployment* stage pattern

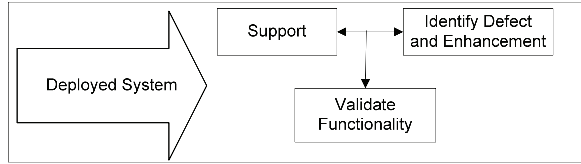

Figure 16. Components of *Maintenance* stage pattern

As mentioned earlier, the process patterns proposed herein can be used as process components in the assembly-based approach of SME. MDA-based methodologies can thus be constructed through assembling these components based on given organizational settings or the characteristics of the project at hand. Furthermore, the paradigm-based approach of SME can use MDASP as a metamodel to instantiate and adapt process- and product models. Processes can thus be built by using a predefined instantiation and assembly procedure using the proposed metamodel and patterns. This approach is very similar to that of OPEN/OPF [15] and Rational Method Composer (RMC) [16].

TABLE I. REALIZATION OF THE PROPOSED PROCESS PATTERNS IN MAJOR MDA-BASED METHODOLOGIES

Methodology	***Phases***	***Corresponding stage process patterns***
MODA-TEL	*Project management phase*	*Justify, Planning and Management*
	Preliminary preparation phase	*Define Infrastructure*
	Detailed preparation phase	*Define Infrastructure*
	Infrastructure setup phase	*Define Infrastructure*
	Execution phase	*Requirements Analysis, Define PIM, Transformation, Coding/Testing, Test in the Large, Deployment, Maintenance*
MASTER	*Capture user requirements phase*	*Requirements Analysis*
	PIM context definition phase	*Define CIM*
	PIM requirements specification phase	*Requirements Analysis*
	PIM analysis phase	*Define PIM*
	Design phase	*Define PIM, Transformation*
	Coding and integration phase	*Coding/Testing*
	Test phase	*Coding/Testing, Test in the Large*
	Deployment phase	*Deployment*
C^3	*Standardization phase*	*Generalization*
	Software development phase	*Define PIM*
	Model design phase	*Define PIM*
	Code generation	*Transformation, Coding/Testing*
	Application deployment phase	*Deployment*
ODAC	*Analysis phase*	*Define PIM*
	Design phase	*Transformation*
	Implementation phase	*Transformation, Coding/Test, Test in the Large*
DREAM	*Domain analysis phase*	*Requirements Analysis*
	Product line scoping phase	*Requirements Analysis*
	Framework modeling phase	*Define PIM*
	Application requirements	*Requirements Analysis*
	Application-specific design phase	*Define PIM*
	Framework instantiation phase	*Define PIM, Transformation*
	Model integration phase	*Define PIM, Transformation*
	Application detailed design	*Transformation*
	Application implementation phase	*Transformation, Coding/Testing*
DRIP-Catalyst	*Problem to solution transition phase*	*Define PIM*
	Platform-independent architectural design phase	*Define PIM*
	Platform-independent detailed design phase	*Define PIM*
	Formal verification phase	*Transformation*
	PIM to PSM transition phase	*Transformation*
	PSM to code phase	*Transformation*
	Completion phase	*Coding/Testing*
	Deployment phase	*Test in the Large, Deployment*

VII. Conclusions and Future Work

We have introduced a set of process patterns and a generic process model for MDA-based software development. The generic process model organizes and structures the patterns in a cohesive lifecycle. The patterns were identified through top-down refinement of the generic process and the study of six well-known MDA-based methodologies. We have shown that the proposed process patterns fully cover the six methodologies used as pattern sources. The resulting patterns can be used in method engineering to build a bespoke MDA-based methodology or to adapt an existing software process to MDA standards.

The research can be furthered in several directions. The finer-grained task process patterns have not been thoroughly covered; a more in-depth analysis is required for providing comprehensive coverage of the task patterns. Due to the lack of sources of process patterns for MDA-based umbrella activities, they have not been addressed in this research; focus can be shifted to heavyweight methodologies and other MDD approaches as sources of insight into umbrella process patterns. A similar research is being conducted on embedded real-time methodologies. The ultimate goal is to exploit the similarities between embedded real-time and MDA-based processes in order to explore the existence of patterns in hardware-software co-design.

References

[1] J. Miller, and J. Mukerji, MDA Guide Version 1.0.1, Object Management Group (OMG), 2003.

[2] S.W. Ambler, Process Patterns: Building Large-Scale Systems Using Object Technology, Cambridge University Press, 1998.

[3] R. Ramsin, and R.F. Paige, “Process-centered review of object oriented software development methodologies,” ACM Computing Surveys, vol. 40, no. 1, 2008, pp. 1-89, doi: 10.1145/1322432.1322435.

[4] P. Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley, 2000.

[5] J. Ralyté, R. Deneckère, and C. Rolland, “Towards a generic model for situational method engineering,” Lecture Notes in Computer Science, vol. 2681, 2003, pp. 95–110, doi: 10.1007/3-540-45017-3.

[6] M. Gnatz, F. Marschall, G. Popp, A. Rausch, and W. Schwerin, “Towards a living software development process based on process patterns,” Lecture Notes in Computer Science, vol. 2077, 2001, pp. 182–202, doi: 10.1007/3-540-45752-6.

[7] A. Gavras, M. Belaunde, L.F. Pires, and J.P. Almeida, “Towards an MDA-based development methodology,” Lecture Notes in Computer Science, vol. 3047, 2004, pp. 230–240, doi: 10.1007/b97879.

[8] X. Larrucea, A.B.G. Dıez, and J.X. Mansell, “Practical model driven development process,” Proc. Second European Workshop on Model Driven Architecture, 2004, doi: 10.1.1.95.5229.

[9] T. Hildenbrand and A. Korthaus, “A model-driven approach to business software engineering,” Proc. Eighth World Multi-Conference on Systemics, Cybernetics and Informatics, 2004, pp. 18–21, doi: 10.1.1.84.8505.

[10] M.P. Gervais, “ODAC: An agent-oriented methodology based on ODP,” Autonomous Agents and Multi-Agent Systems, vol. 7, no. 3, 2003, pp. 199–228, doi: 10.1023/A:1024797300606.

[11] S.D. Kim, H.G. Min, J.S. Her, and S.H. Chang, “DREAM: A practical product line engineering using model driven architecture,” Proc. Third International Conference on Information Technology and Applications, 2005, pp. 70–75, doi: 10.1109/ICITA.2005.118.

[12] N. Guelfi, R. Razavi, A. Romanovsky, and S. Vandenberg, “DRIP Catalyst: an MDE/MDA method for fault-tolerant distributed software families development,” Proc. OOPSLA & GPCE Workshop on Best Practices for Model Driven Development, 2004.

[13] S. Tasharofi, and R. Ramsin, “Process patterns for agile methodologies,” International Federation for Information Processing-IFIP, Vol. 244, 2007, pp. 222-237, doi: 10.1007/978-0-387-73947-2.

[14] DSDM Consortium, and J. Stapleton, DSDM: Business-Focused Development, 2nd ed., Addison-Wesley, 2003.

[15] D. Firesmith, and B. Henderson-Sellers, The OPEN Process Framework: An Introduction, Addison-Wesley, 2001.

[16] P. Kroll, “Introducing IBM Rational Method Composer,” Nov. 2005, published on the Web at: http://www-128.ibm.com/developerworks/rational/library/nov05/kroll.