Meduse: an Approach for Tailoring Software Development Process

Sara Casare, Tewfik Ziadi, Anarosa a F Brandão, Zahia Guessoum

Meduse: an Approach for Tailoring Software Development Process

Sara Casare[1], Tewfik Ziadi[2]
[1] LTI, University of São Paulo, Brazil
[2] LIP6, University Paris-Sorbonne, France
[4] CReSTIC, University Reims, France

Anarosa A. F. Brandão[2,3], Zahia Guessoum[2,4]
[3] Computing Engineering and Digital Systems Department
University of São Paulo, Brazil

***Abstract*—**Software processes, as software products, are variable across projects and thus a one-size-fits-all approach does not work out for development processes. We propose Meduse, an approach for tailoring development processes according to project needs. Such an approach, which is based on software product line and method engineering techniques, takes into account processes similarities (i.e. commonalities) and differences (i.e. variabilities), as well as reusable process fragments. Having processes tailored on demand according to the project needs shall reduce project risks, rise best practices adoption by the development team, support project planning and budget managing, among other benefits.

Keywords *- Process Tailoring; Software Product Line; Software Process Line; Method Engineering.*

I. INTRODUCTION

Software Processes aim at improving the quality and productivity of software development by encoding sets of well-known practices for realizing them. Several modelling languages and tools have been proposed to represent and manage software process in the Method Engineering field [1][2]. Among them we find the *Software and System Process Engineering Meta-model* (SPEM) [3], which is the *de facto* standard for modelling and managing processes. Fig. 1 illustrates an example of a software process to develop agile projects based on Extreme Programming (XP) [4] described using SPEM standard notations. This software process defines three ordered phases: *Write Story Phase*, *Write Code Phase* and *Integration Test Phase*. Each phase comprises a set of activities. For instance, the *Write Code Phase* contains three activities related to code writing.

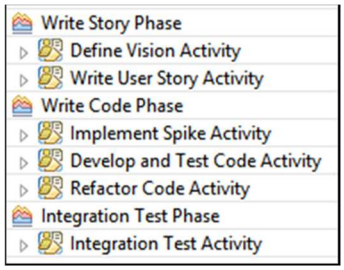

Fig. 1. An example of a software process to develop agile projects

Specifying software processes using standard modelling languages allows capitalizing the best practices from companies throughout different projects. However, as underlined by Rombach [5] and Brinkkemper [1] among others, a software development process should not be used before being tailored according to current project needs. Otherwise, the project risks wasting work already done and producing artifacts of little added value. To illustrate this, let us consider the example of the software process of Fig. 1. Indeed, in some agile projects the *Write Story Phase* can be omitted whenever the requirement specification phase is skipped. In addition, refactoring is not a mandatory activity inside the coding phase. Therefore, this software process should be tailored according to some variability factors that are related to the specificities of each project.

One promising approach of achieving process tailoring is what is designed as Software Process Lines [5][6]. Software Process Line Engineering (SPrLE) aims to organize a family of processes according to their similarities (i.e. commonalities) and differences (i.e. variabilities) in order to achieve a better process tailoring according to a specific project needs. SPrLE reuses concepts proposed by Software Product Line Engineering (SPLE) [7][8]. The basic idea is to apply concepts of product lines to the domain of software process models. This mainly includes two main principles: i) mechanisms to explicitly specify variability in software processes, and ii) mechanisms to tailor (also called derive) specific process variants according to each project needs.

Many SPrLE approaches have been proposed in last years [9][10][11]. However two limitations can be clearly identified, being the first concerned to variability specification. Indeed, most existing approaches consider variability at fine-grained process elements in software processes, like work products, roles, and tasks. In this paper, we advocate the idea that variability should be specified in terms of user needs at the domain level independently from the process elements that are more related to the implementation details. Besides, the general framework proposed by the SPLE community [8] suggests to specify the variability at the problem space side that is related to domain analysis, which is independent from the solution space side that is related to domain implementation. The second limitation concerns the tailoring of the process variants. While variability is only specified at fine-grained process elements, tailoring software process according to specific needs of a project requires to explicitly reasoning on these several fine-grained process elements to identify the process elements that map the project needs. We believe that this practice is by no means of the principal of product derivation in SPLE. Indeed, product derivation should be guided by a reasoning that is based on the user needs that are defined at a high level of abstraction and not at the fine-grained elements like the tasks and roles of the implementation side.

In order to overcome such limitations this paper proposes Meduse, a new SPrLE approach to tailor software processes according to project needs that tackles process tailoring as a process variant management problem, as presented in the remaining sections.

II. MEDUSE APPROACH OVERVIEW

Meduse proposes developing process lines based on both SPLE and Method Engineering principles. On one hand, Meduse takes advantages of SPLE principles and techniques to manage process variability at a high level of abstraction, as well as to automatically derive valid process variants. On the other hand, it adopts Method Engineering principles to build reusable process artifacts, called process fragments, to be linked to the process variability model.

Meduse follows the general SPLE framework proposed by Apel et al.[8], and encompasses four phases: domain analysis, domain implementation, requirement analysis, and variant implementation. Fig. 2 presents the big picture of the Meduse approach, showing both process line artifacts and the final process variant that is automatically generated according to project needs. First, during domain analysis we propose to represent the process domain knowledge in terms of commonalities and variabilities among members of a same process line by means of a popular SPLE artifact, a Feature Model [12]. Second, to deal with domain implementation we propose using reusable process fragments together with a compositional approach based on the Pure Delta-oriented programming [13], in which process derivation relies on the application of process delta modules over an empty process: the modification proposed by a process delta module consists of the addition and/or removals of reusable process fragments from the process variant. By using process delta modules we can define the partially-ordered sequence in which process fragments will appear in the process variant, since process delta modules are grouped in fixed-ordered partitions, and modules in the same partition can be applied in any order, as suggested in [13]. Therefore, by adopting the Pure Delta-oriented programming our approach covers an important issue of software process derivation: software processes are represented as a partially-ordered sequence of work. Third, process variants are specified through feature selection according to project needs using another popular SPLE artifact, a Feature Configuration [12]. Finally, we propose a tool – the Meduse Composer - that automatically derives the specified process variant: it takes the empty process as starting point of the work breakdown structure and incrementally adds or removes process fragments according to the modifications specified in the process delta modules connected with the selected features. In the following we present the proposed approach, starting with the Process Domain Analysis.

III. PROCESS DOMAIN ANALYSIS

Based on Kang et al. [12] we define a **Process Feature** as a distinctively identifiable process characteristic required by the process users. On one hand, process features are used to communicate process capabilities at a high level of abstraction. On the other hand, they establish how members of a process line can vary. Process features are organized in a feature model that describes relationships between them, and formally specifies which feature selections are valid. This is made by hierarchically representing features in a diagram, where edges are used to represent parent-child relationships between them (see Fig. 3), as suggested in [12].

We may use process features to represent the major areas of concern that a process line shall cover. Thus, they may represent software technical disciplines, like requirement, analysis, design, implementation, and test disciplines, as well as software development practices covered by the process line, as code standards and pair programming. Last but not least, process features may be used to show how project management aspects may vary among the family members. Fig. 3 depicts an example of a feature diagram containing four features: *Implementation*, *Testing*, *UnitTest*, and *IntegrationTest*. Such a diagram shows that we shall have family members including Implementation and/or Testing disciplines, but whenever we have a process variant containing Testing, UnitTest is mandatory, while IntegrationTest is an optional feature.

One of the main advantages of using feature models to represent process variability is that we can use a Satisfiability (SAT) solver tool to determine whether a feature model is consistent, i.e. we have at least one valid process configuration, and whether or not a given feature configuration is valid, and therefore it will give rise to a valid process variant.

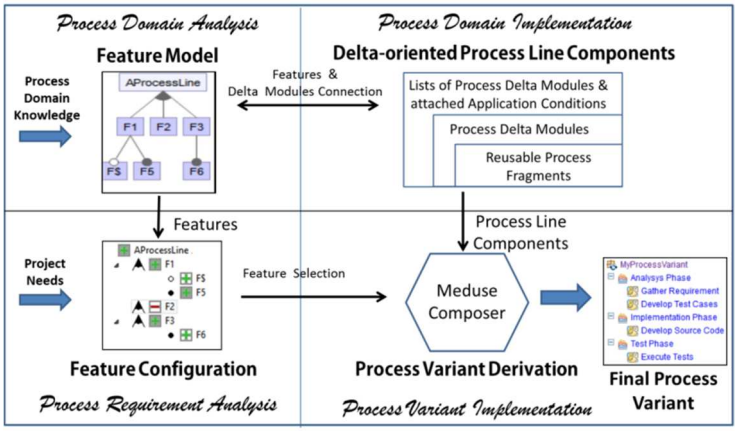

Fig. 2. A big picture of Meduse approach

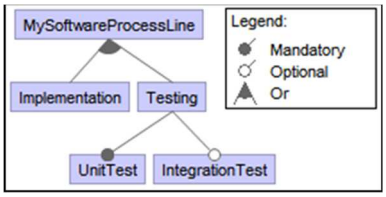

Fig. 3. An example of a Feature Diagram for a process line

IV. PROCESS DOMAIN IMPLEMENTATION

While feature models allow representing features in the domain analysis side, on the implementation side Meduse adopts Pure Delta-oriented programming to deal with process variabilities. By adopting this approach, a Meduse process line comprises a list of process delta modules connected to features through applications conditions, and a set of reusable process fragments embedded in these deltas, as described in the remainder of this section.

A. Reusable Process Fragment

Based on the Method Engineering notions [1] [2], we define a **Process Fragment** as a reusable process building block that represents a portion of some work breakdown structure of a software development process in terms of process meta-model elements. In order to form such a work breakdown structure, a process fragment may contain iterations, phases and activities, which in their turn may encompass task(s) that usually produces work product(s), and is performed by development role(s). Fig. 4 shows an example of process fragment using SPEM as the underlying process meta-model: the *Execute Unit Test Activity* encompasses two tasks, *Create Unit Test Cases* and *Run Unit Tests*. These tasks are performed by a *Developer* and produce *Test Cases* and *Tested Source Code* as work products.

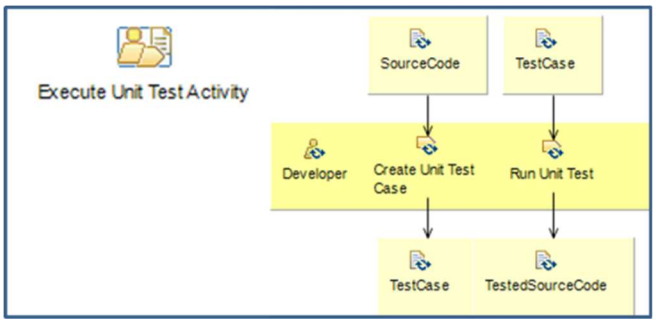

Fig. 4. A process fragment for Unit Test

B. Process Delta Module

Based on the Pure Delta-oriented Programming approach, we define a **Process Delta Module** as a container that encompasses modifications to processes in terms of additions and/or removals of process fragments. Therefore, a delta module allows to add or remove portions of reusable process breakdown structures previously specified as reusable process fragments.

Following our example, Fig. 5 depicts two process delta modules: *DevCode* and *TestU*. The former proposes the addition of the *Develop Source Code* process fragment, while the latter proposes the addition of *Set Test Environment* and *Execute Unit Test* fragments.

```
Process Delta Module DevCode {
        adds process fragment Develop Source Code
}
Process Delta Module TestU {
        adds process fragments Set Test Environment,
                               Execute Unit Test
}
```

Fig. 5. Two process delta modules for dealing with coding and testing

C. List of Process Delta Modules and Application Conditions

We adopted application conditions [13] to create the connection between the modifications prescribed in process delta modules and the process features. In Meduse, an **Application Condition** is a propositional formula attached to every process delta module through a *when* clause. It determines for which features the specified modifications are to be carried out. Examples of application conditions are presented in Fig. 6. The list formed by all process delta modules and attached application conditions determines the modifications required to derive process variants, and the order in which such delta modules must be applied during process derivation.

Moreover, process delta modules are grouped in partitions, which are enclosed by brackets (e.g. […….]). Process delta modules in the same partition may be applied in any order. However, the order of the set of partitions is fixed. On one hand, the fixed partition order can be used to generate processes that require a total ordered work breakdown structure. On the other hand, partial order inside a partition can be used to generate processes that require work break structures containing group of activities that may be executed in any order. Fig. 6 illustrates a list of four process delta modules attached to application conditions involving features presented in Fig. 3. In this example, the process delta module *DevCode* shall be applied whenever the feature configuration contains the *Implementation* feature, *TestU* whenever it contains *UnitTest* feature, *TestI* whenever *IntegrationTest* feature is present, and finally *Deploy* shall be applied whenever we have both *Implementation* and *IntegrationTest* features. Additionally, these modules are defined in distinct partitions and thus this list establishes that all derived process variants would follow a total ordered work breakdown structure.

```
List of Process Deltas & Application Conditions {
[ DevCode   when Implementation ]
[ TestU     when UnitTest]
[ TestI     when IntegrationTest ]
[ Deploy    when Implementation AND IntegrationTest]
}
```

Fig. 6. A list of Process Deltas and Application Conditions

V. REQUIREMENT ANALYSIS AND VARIANT DERIVATION

The selection of a set of features for a particular project is made through a *feature configuration*. Fig. 7 shows a valid feature configuration for the feature model previously presented (see Fig. 3), where *Implementation*, *Testing* and

UnitTest form the set of selected features, which excludes *IntegrationTest* feature.

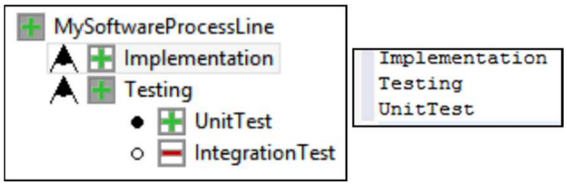

Fig. 7. A valid feature configuration

Process variant derivation consists of incrementally applying to an empty process the modifications specified by the process delta modules connected with valid application conditions. By a valid application condition we mean an application condition that consists of a propositional formula that is evaluated to true for the given feature configuration [13].

The derivation of a process variant for a given feature configuration is achieved according to the following steps, implemented by the Meduse Composer: (i) Find all process delta modules that shall be applied to the process variant, i.e., those modules attached to an application condition evaluated to true for the given feature configuration. For example, whenever a feature configuration encompasses the features *Implementation* and *Testing*, the process delta modules *DevCode* and *TestU* shall be selected (ii) Generate the process variant by applying the modification proposed by the selected process delta modules respecting the total order on the partitioning of process delta modules. Note that the first process delta is applied to the empty process and thus it must start by adding process fragments, and not removing them.

Following our example, modifications proposed by *DevCode* and *TestU* modules are applied to the process variant, by adding the process fragments *Develop Source Code*, *Set Test Environment* and *Execute Unit Test*. Fig. 8 depicts a process variant derived according to the feature configuration of our example. Such a process encompasses the *Building Cycle Phase* that comprises three activities for dealing with source code development and unit testing. Moreover, Fig. 8 offers an expanded view of the last activity, showing its two tasks: *Create Unit Test Case* and *Run Unit Test* tasks.

Fig. 8. A process variant derived according to a feature configuration

VI. IMPLEMENTATION

We have developed a prototype implementation of Meduse, in which we adopted FeatureIDE [14] as tool to specify the feature model and to create feature configurations, as well as SPEM and Eclipse Process Framework[1] to manage process fragments and delta modules. Moreover, the Meduse Composer was developed and fully integrated as a new composer within the FeatureIDE to automatically derive process variant according to a given feature configuration, ensuring the partially-ordered sequence of process fragments specified in the process line. Interested readers may access the Meduse website[2] to see process variants automatically derived by this prototype.

VII. CONCLUSION

This paper proposed Meduse, an approach to automatically tailor software processes according to project needs. Meduse follows the general approach for SPLE combined with Method Engineering principles. Contrary to existing SPrLE approaches [9][10][11], Meduse is built by a rigorous respect of SPLE principles: process variability is specified in terms of project needs at a high level of abstraction (at the problem space), while process derivation (at the problem solution space) is guided by a reasoning based on these needs, independently from the process elements used in the SPrLE implementation.

ACKNOWLEDGEMENT

Sara Casare was supported by CNPq (grant #233828/2014-1), Brazil.

REFERENCES

[1] S. Brinkkemper, "Method Engineering: Engineering of Information Systems Development Methods and Tools," Information and Software Technology, vol. 38 (4), pp. 275-280, 1996.

[2] A.F. Harmsen, Situational Method Engineering, M. E. & Young, 1997.

[3] OMG. Object Management Group, "Software & Systems Process Engineering Meta-Model Specification," version 2.0, OMG document number: formal/2008-04-01, 2008.

[4] K. Beck, "Embracing change with extreme programming," Computer, vol. 32(10), pp.70-77, 1999.

[5] D. Rombach, "Integrated Software Process and Product Lines," in: Li, M., Boehm, B., Osterweil, L., (Eds.) ISPW, Beijing, China: Springer, pp. 83-90, 2005.

[6] S. M. Sutton Jr, and L. J. Osterweil, "Product families and process families," in Software Process Workshop, PSSPL, IEEE, 1996.

[7] P. Clements, and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley Longman Publishing Co. Inc., Boston, 2001.

[8] S. Apel, D. Batory, C. Kästner, and G. Saake, "Feature-Oriented Software Product Lines," Berlin: Springer, 2013.

[9] J. Simmonds, M. Bastarrica, L. Silvestre, and A. Quispe, "Analyzing methodologies and tools for specifying variability in software processes," TR/DCC-2011-12, Univer. de Chile, 2011.

[10] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J. M. Jézéquel, "Leveraging CVL to manage variability in software process lines," in APSEC 2012, vol. 1, pp. 148-157, IEEE, 2012.

[11] J. A. Hurtado, M. C. Bastarrica, S. F. Ochoa, and J. Simmonds, "MDE software process lines in small companies," Journal of Systems and Software, vol. 86(5), pp. 1153-1171, 2013.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," (No. CMU/SEI-90-TR-21), Carnegie-Mellon Univ, 1990.

[13] I. Schaefer, and F. Damiani, "Pure delta-oriented programming," in Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD), pp. 49-56, ACM, 2011.

[14] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T Leich, "FeatureIDE: An extensible framework for feature-oriented software development," Science of Computer Program., vol. 79, pp.70-85, 2014.

[1] https://eclipse.org/epf/

[2] https://pages.lip6.fr/Tewfik.Ziadi/iceccs16/