Integrating Software Process Reuse and Automation

Emmanuelle Rouillé, Benoit Combemale, Olivier Barais, Touzet David, Jean-Marc Jézéquel

Integrating Software Process Reuse and Automation

Emmanuelle Rouillé*†, Benoît Combemale†, Olivier Barais†, David Touzet* and Jean-Marc Jézéquel†
*Sodifrance
P.A. la Bretèche, avenue Saint-Vincent, 35768, Saint-Grégoire, France
Email: {erouille, dtouzet}@sodifrance.fr
†Université de Rennes 1
IRISA, Campus de Beaulieu, 35042, Rennes, France
Email: {emmanuelle.rouille, benoit.combemale, olivier.barais, jean-marc.jezequel}@irisa.fr

***Abstract*—Reusing software processes from a Software Process Line (SPL, *i.e.,* a set of software processes that captures their commonalities and variabilities) and automating their execution is a way to reduce development costs. However, to our best knowledge no approach integrates both aspects. The difficulty is to automate the execution of a process whose variability is only partially resolved (*i.e.,* a value is not set to each variable part of the process). Indeed, according to projects' constraints, it is possible to start the execution of a part of a process whose variability is resolved, while postponing the resolution of the variability of other parts of this process. In this paper, we propose a tool-supported approach that integrates both aspects. It consists of reusing processes from an SPL according to projects' requirements. The processes are bound to components that automate their execution. When the variability of a process to execute is not fully resolved, our approach consists of resolving this variability during the execution of this process. We illustrate this work on a family of processes for designing and implementing modeling languages. Our approach enables both the reuse of software processes and the automation of their execution, while enabling to resolve process variability during the execution.**

Keywords*—*Software Process; Product Line; Automation

I. Introduction

A software development process defines the sequence of steps to perform in order to transform customer requirements into a software [1]. Reusing software processes and automating their execution when possible both aim at reducing software development costs. Indeed, reusing software processes enables to reuse the know-how that software engineering companies acquired during projects. Therefore, this prevents people that intervene in new software engineering projects from doing the same errors again. This also provides best practices to these people. As for the automation of the execution of software processes, it prevents people from manually performing recurrent tasks, which is time consuming and error prone.

The difficulty in reusing software processes lies in the fact that the variability of the requirements of the projects often implies variability of software processes. Therefore, existing approaches that aim at reusing software processes rely on the management of the variability of the software processes [2]–[6], using Software Process Line Engineering (SPLE) [7]. These approaches capture a family of software processes into a Software Process Line (SPL), that specifies commonalities and variabilities between these processes [3]. These approaches rely on Model-Driven Engineering (MDE) to define the SPL. A software process of the SPL can be reused by deriving it from the SPL, according to the requirements of a project. Deriving a process from the SPL consists of resolving the variability of the SPL, *i.e.,* setting a value to each variable part of the SPL. Other approaches aim at automating the execution of software processes (*e.g.,* [8]). They consist of automating the execution of the activities of a process when it is possible. However, to our knowledge none of these approaches integrates at a time the reuse of software processes from an SPL and the automation of their execution, whereas this would enable to go further in the reduction of development costs.

Automating the execution of software processes reused from an SPL is not a trivial task. Indeed, it may be needed to execute a software process derived from an SPL, while the variability is only partially resolved. For instance, in order to save time it is possible to start the execution of a Java development process (*e.g.,* by writing the functional specifications) whereas the GUI framework (*e.g.,* Struts, JSF, Flex or GWT) is not chosen yet. In this case, the process execution may be erroneous due to the unresolved variability. Therefore, to automate the execution of a process reused from an SPL, it is important to handle the cases where the variability of a process under execution is only partially resolved.

In this article, we address this need by proposing a tool-supported approach that integrates the reuse of software processes and the automation of their execution. Our approach consists of defining an SPL and of reusing processes from this SPL according to the requirements of the projects. The software processes of the SPL are bound to components that automate manual recurrent tasks that occur during their execution. When the variability of a process under execution is not fully resolved, our approach consists of resolving this variability during the execution. To reuse the software processes from an SPL, we rely on a previous work [6] that consists of using the Common Variability Language (CVL)[1] [9] to manage the variability of software processes. To bridge the gap between the software processes and the components that automate them, we rely on another previous work [10] that consists of binding the components to the SPL in order to identify their contexts of use. This helps to implement components that are reusable across their different contexts of use. In this article, we go further by detailing how to automate the execution of software processes derived from an SPL, even if their variability is partially resolved. We illustrate all along the paper our work on a family of processes for designing and implementing modeling languages (*i.e.,* metamodeling processes).

[1] http://www.omgwiki.org/variability/

This paper is organized as follows. Section II presents the process modeling language SPEM 2.0 [11] and CVL. Section III presents the illustrative example we use throughout this article. We present our approach in Section IV and its proof-of-concept tooling support in Section V. We discuss them in Section VI. The related work is discussed in Section VII. We conclude and present our perspectives of work in Section VIII.

II. Background

In this section we present SPEM and CVL. Indeed, in this article we use SPEM as a process modeling language and CVL as a language for specifying and resolving variability. It is possible to use other languages according to one's needs.

A. SPEM 2.0

We only introduce the subset of SPEM 2.0 required to understand the following of this article. The SPEM 2.0 specification provides the concepts of task ( ), work product ( ), role ( ) and tool ( ). A task is a work to realize during the process execution. Tasks take zero or several work products as inputs and outputs. Zero or several roles perform a task. Zero or several tools support the realization of a task.

In order to model the flow of tasks, we use the concepts of control flow ([guard]), initial node ( ), final node ( ), decision node ( ), fork node ( ) and join node ( ) from UML 2 [12] activity diagrams.

B. CVL

CVL is a domain-independent language for specifying and resolving variability over any instance of any MOF-compliant metamodel (*i.e.,* a base model). CVL contains several layers as illustrated by the black part of Fig. 1.

The Variability Abstraction Model (*VAM*) is in charge of expressing the variability in terms of a tree-based structure. The core concept of the VAM is the variability specification (*VSpec*). The variability specifications are nodes of the VAM. One kind of variability specification is the choice (*Choice*). Choices can be intuitively compared to features. A choice can or cannot be selected during the product derivation whether it is resolved to true or not. CVL also provides the concept of derived variability specification, whose resolution is derived from the resolution of other variability specifications. A derivation condition specifies how to derive the resolution of a derived variability specification according to the resolution of other variability specifications.

Besides the VAM, CVL also contains a Variability Realization Model (VRM). This model provides a binding between the base model and the VAM. It makes possible to specify the changes in the base model implied by the resolutions of the variability specifications. These changes are expressed as variation points (*VariationPoint*) in the VRM. The variation points capture the derivation semantics, *i.e.,* the actions to perform on the base model during the derivation. A variation point is bound to a set of variability specifications. A variation point is performed when all its bound variability specifications are resolved to true.

Finally, CVL contains Resolution Models (RM) to resolve the variability captured in the VAM. The core concept of the RM is the variability specification resolution (*VSpecResolution*), that resolves a variability specification.

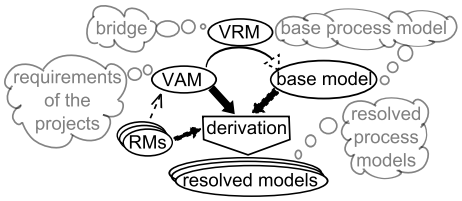

Figure 1: CVL overview (in black) and its use with software processes (in gray)

A resolved model can be derived from the base model, according to the VAM, the VRM and a given RM. The resolved model thus corresponds to the base model, in which variability has been resolved.

III. Illustrative Example

We present a simplified metamodeling process as an illustrative example. A metamodeling process consists of designing and implementing a modeling language (including editor, compiler, etc.). This illustrative example highlights the variability of the process as well as the manual recurrent tasks that occur.

In our example, a metamodeling process is iterative and starts with the definition of a metamodel, which defines the concepts of a business domain and the relationships between them. Optional tasks follow according to the requirements: definition of a textual or tree editor, a checker, an interpreter and a compiler. The tool used to define the metamodel and the tree editor is EMF[2] (Eclipse Modeling Framework). The tool used to define the textual editor varies according to the requirements of the projects: in our example it is either XText[3] or EMFText[4]. The tool used to define the checker, the interpreter and the compiler is Kermeta[5], a metamodeling environment. The tool used to put the interpreter under version control also varies according to the requirements of the projects: here it is either the SVN Version Control System (VCS) or the Git one.

These tasks require manual recurrent interventions that would benefit to be automated. For instance, all the tasks except the tree editor definition start with the creation and initialization of a project the first time they occur. For instance, the interpreter definition starts with the creation of a Kermeta project. Furthermore, in our case the implementation of the interpreter respects a specific design pattern, which is the interpreter design pattern. Therefore, after the creation of a Kermeta project, the beginning of the interpreter definition consists of generating the interpreter design pattern for each metaclass of the metamodel. The following times that the interpreter definition occurs, it starts with the generation of the interpreter design pattern only for the new metaclasses of the metamodel. Furthermore, the interpreter definition always ends with its putting under version control the first time it occurs and with the commit of its modifications on a shared repository the following times. The metamodel definition ends each time

[2] http://www.eclipse.org/modeling/emf/
[3] http://www.eclipse.org/Xtext/
[4] http://www.emftext.org
[5] http://www.kermeta.org/

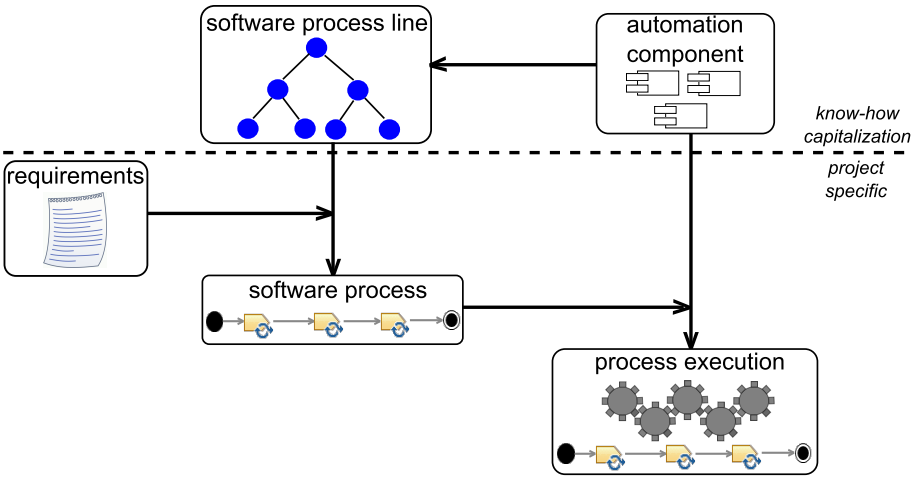

Figure 2: Overview of our approach to reuse software processes and automate their execution

it occurs with the validation of the produced metamodel. The first time it occurs, the tree editor definition always starts with the creation of a file that will contain the concrete syntax of the metamodel. The following times the tree editor definition always starts with the update of this file. The editor definitions end each time they occur with the generation of the editors themselves. Other manual recurrent interventions, that do not apply only to metamodeling processes, also occur, like the configuration of the development environment, of the continuous integration environment and of the build schema.

IV. Approach

In this section we present our approach that integrates the reuse of the software processes and the automation of their execution. We start by giving an overview of this approach. We then explain how we realize the reuse of the software processes. After that we detail how to define components that automate manual recurrent tasks that occur during the process execution (*i.e.,* automation components, abbreviated as ACs). We finish by presenting how to automate the process execution.

A. Approach Overview

Figure 2 shows an overview of our approach to integrate the reuse of software processes and the automation of their execution. Its principle is to capture a family of software processes into an SPL. ACs are bound to the work units (*e.g.,* tasks, activities...) they automate in the SPL. It is possible to derive a process from the SPL according to the requirements of a project and to execute this process. During the execution of this process, it is possible to automatically launch the execution of the ACs bound to this process. Indeed, the link between the ACs and the work units they automate enables to know when to launch the ACs during the execution of the process. Launching the execution of ACs during the execution of a process enables to automate the execution of this process.

B. Reusing the Software Processes

We now detail how we implement in our approach the reuse of software processes. In a previous paper [6], we use CVL to define an SPL and to automatically derive a process from this SPL according to the requirements of a project. The gray part of Fig. 1 shows an overview of this approach.

The base model is a base process model that contains the process elements required to define the expected family of processes. Fig. 3 shows the base process model of the illustrative example.

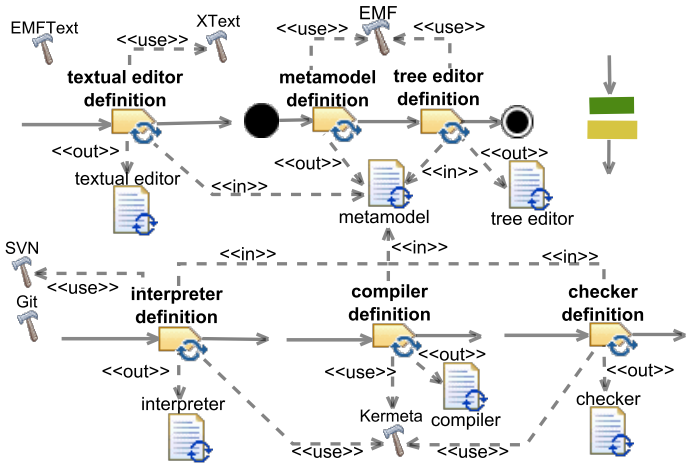

Figure 3: Base process model of the illustrative example

The VAM specifies the variability of the requirements of the projects. Fig. 4 shows the VAM of the illustrative example. It specifies that requirements are optional (*i.e., interpreter*, *textual editor*, ...), that the textual editor definition requires either XText or EMFText and that a VCS is mandatory, which is either SVN or Git. There are also derived choices (*i.e., parallelization*, *parallel interpreter*, *parallel tree editor*...). They do not represent any requirements and we will see in the following paragraph that they are only useful to define the VRM. The derived choice named *parallelization* is automatically resolved to true when at least two optional choices among *interpreter*, *checker*, *compiler*, *tree editor* and *textual editor* are resolved to true. The goal of the other derived choices is to know if the optional tasks of the metamodeling process will be in sequence or in parallel. For instance, if the derived choice named *parallel interpreter* is resolved to true, then it means that the optional task named *interpreter definition* will be part

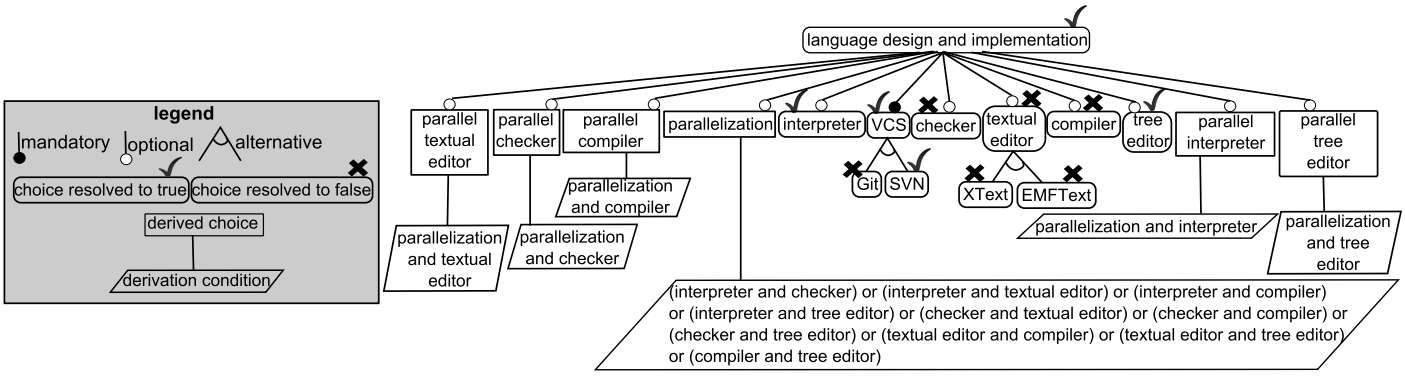

Figure 4: VAM and an RM of the illustrative example

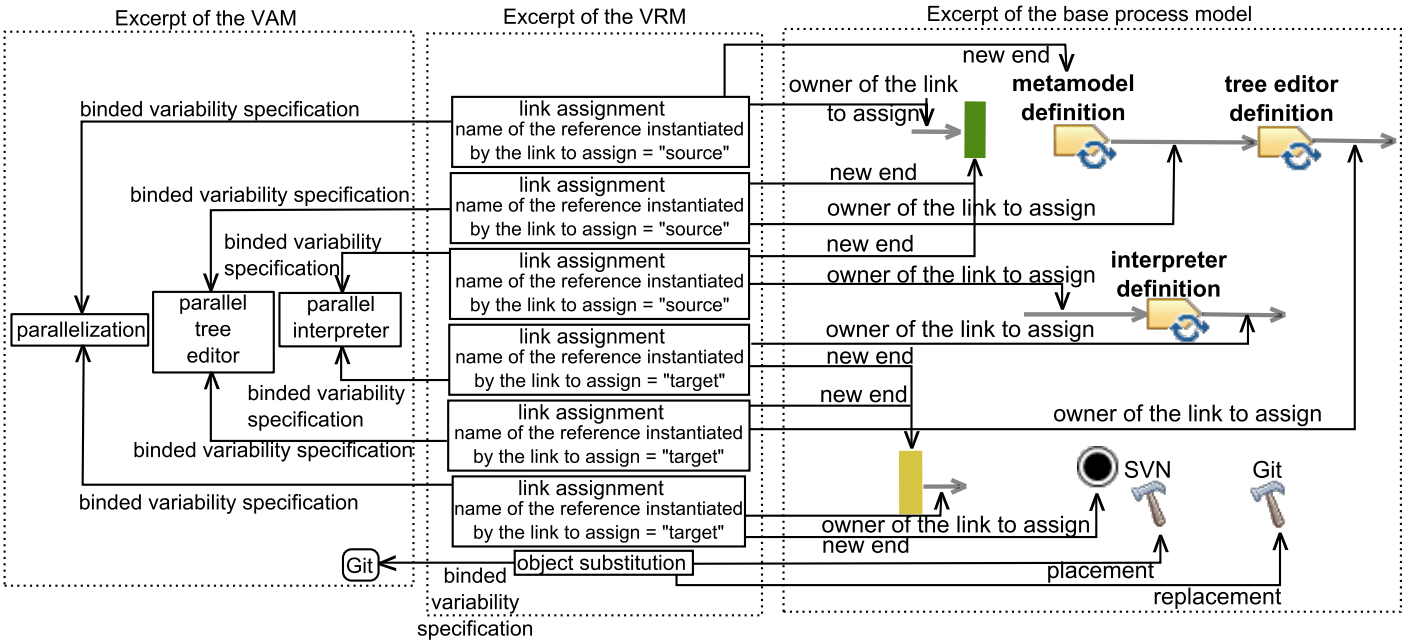

Figure 5: Excerpt of the VRM of the illustrative example

of the metamodeling process and will be performed in parallel.

The VRM defines which process elements of the base process model are impacted by a requirement of the VAM and how these process elements are impacted. For a sake of place and clarity, the center part of Fig. 5 only shows an excerpt of the VRM of the illustrative example. The right part of Fig. 5 is an excerpt of the base process model of Fig. 3 and the left part of Fig. 5 is an excerpt of the VAM of Fig. 4. Here the VRM specifies that if Git is the VCS, then it must replace SVN. If the derived choice named *parallelization* is resolved to true, then the source of the control flow entering the fork node must be assigned to the task named *metamodel definition* and the target of the control flow coming from the join node must be assigned to the end node. If the derived choice named *parallel tree editor* is resolved to true, then the source of the control flow entering the task named *tree editor definition* must be assigned to the fork node and the target of the control flow coming from the same task must be assigned to the join node. Similar operations are applied to the control flows entering and coming from the task named *interpreter definition* when the derived choice named *parallel interpreter* is resolved to true.

The RM specifies the resolution of the variability of the requirements. Fig. 4 shows an RM for the illustrative example (represented by ✓ and ✗). Only the choices named *interpreter* and *tree editor* are resolved to true and SVN is the VCS. The derived choices are automatically resolved according to the resolution of the choices. Here only the derived choices named *parallelization*, *parallel interpreter* and *parallel tree editor* are resolved to true.

The resolved model is a resolved process model. Fig. 6 shows the resolved process model corresponding to the RM of Fig. 4. It is a metamodeling process with a metamodel definition, a tree editor definition, an interpreter definition and with SVN as VCS.

C. Defining the Automation Components

We now detail how to define ACs. The difficulty in defining ACs is that a unique AC can automate several work units in a same process or across different processes. Therefore, an AC must be reusable across its different contexts of use. These different contexts of use are represented by the different work units an AC contribute to automate. In a previous work [10],

we present a methodology that aims at implementing ACs that are reusable across their different contexts of use. We give an overview of this methodology in the following paragraph.

The overall principle of our methodology is to bind the ACs to the work units of the SPL they automate. This binding enables to identify the different contexts of use of each AC, which is a necessary input to be able to implement ACs that are reusable across their different contexts of use. Indeed, by thinking about how to implement an AC in order to cover all its contexts of use, it is possible to determine the parts of the AC that vary. Mechanisms like for instance parameterization or modularization then enable to implement reusable ACs. If a work unit varies, the binding must specify which variant of this work unit an AC automates. A work unit varies either if this is this work unit itself that varies or if they are the model elements directly related to this work unit (*e.g.,* tools, roles, work products, control flows) that vary.

D. Automating the Process Execution

In order to automate the execution of a process derived from the SPL, it is necessary to know which ACs to execute and when. This is the binding between the ACs and the work unit they automate that provides this information. Indeed, this binding specifies if the execution of an AC occurs during the initialization, execution and/or finalization of a work unit. Furthermore, if several ACs automate a same work unit, the binding specifies in which order these ACs must execute.

On the other hand, the automation of the process execution must handle the fact that a process derived from the SPL may have unresolved variability. Therefore, before the execution of a process part that has unresolved variability, the current actor of the process must resolve this variability.

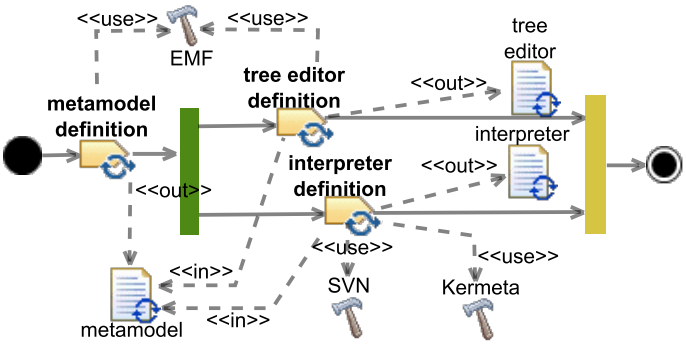

Figure 6: Resolved process model

V. Tooling Support

We now present the tooling we have developed to reuse the software processes, to define the ACs and bind them to the SPL and to automate the execution of a process derived from the SPL. A video demonstration of this tooling is available at http://youtu.be/MYKspY2hE1o. There may be differences between the models of this demonstration and the ones of this article. Indeed, we have simplified the models of this article for a sake of place and clarity. However, the underlying principle is exactly the same. Therefore, the differences between the models of this article and the ones of the video demonstration must not disturb the understanding of the reader.

A. Reusing the Software Processes

We have developed several tools that support the implementation of our approach for reusing software processes (see Section IV-B).

First, we have used EMF in order to generate tree editors to create VAMs and VRMs.

We have also developed a requirement elicitation wizard in order to create RMs. The requirement elicitation wizard is a tool that parses a VAM. For each variability specification of this VAM, the requirement elicitation wizard asks an end user to resolve the variability, through a user-friendly interface. The requirement elicitation wizard creates into an RM the corresponding variability specification resolutions. The end user also has the possibility to skip the variability resolution for each variability specification. In this case, the requirement elicitation wizard does not create any variability specification resolution related to the variability specification whose resolution has been skipped.

Finally, we have developed a CVL derivation engine, that derives a new process model (*i.e.,* a resolved process model) from the base process model, according to the requirements selected in an RM. For each variability specification resolution of an RM, if this variability specification resolution resolves a variability specification to true, then the CVL derivation engine finds the corresponding variability specification into the VAM. Then, the CVL derivation engine performs the variation points that are related to this variability specification. The CVL derivation engine performs the variation points on a copy of the base process model, in order to create the resolved process model without erasing the base process model.

B. Binding the Automation Components and the SPL

We have developed a binding modeler to bind the ACs to the work units of the SPL they automate. Fig. 7 shows the underlying metamodel, namely the binding metamodel. An automated work unit (*AutoWU*) is a work unit automated by a list of ACs (*ACL*). The references named *onStart*, *onDo* and *onEnd* between a list of ACs and an automated work unit specify that the list of ACs automates the initialization, execution and finalization of the automated work unit respectively. A list of ACs defines a set of ACs (*AC*) to execute and their order of execution, through the ordered reference named *ACs*. An automated work unit references a work unit of the SPL through a work unit handle (*WUHandle*) and its attribute named *MOFRef*, which corresponds to the URI of the work unit of the SPL. If a work unit varies, a variant condition (*VariantCond*) (*i.e.,* a condition (*Condition*) expressed with an OCL [13] expressions (*OCLExpr*)) specifies to which variant of the work unit an automated work unit corresponds. Preconditions (*PreCond*) (resp. postconditions (*PostCond*)) specify the required context to execute an AC (resp. the expected results of the execution of the AC).

Fig. 8 shows an excerpt of the binding model of the illustrative example. There are two automated work units (*NewInterpGit* and *NewInterpSVN*), that are two variants of the interpreter definition task (handled by the work unit handle named *InterpDef*). The variant condition named *NoInterpGit* (resp. *NoInterpSVN*) specifies that the automated work unit

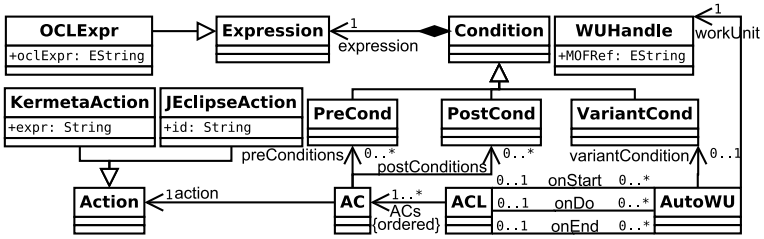

Figure 7: The binding metamodel

named *NewInterpGit* (resp. *NewInterpSVN*) corresponds to the interpreter definition the first time it occurs and with Git (resp. SVN) as VCS. The first time the interpreter definition occurs, the list of ACs named *InitInterp* initializes it by creating a Kermeta project (AC named *NewKermetaProject*) and by generating the interpreter design pattern for each metaclass of the metamodel (AC named *GeneInterp*). The first time the interpreter definition occurs, its finalization consists of its putting under version control. The list of ACs named *EndInterpGit* (resp. *EndInterpSVN*) performs the putting under version control through the AC named *VersionProjectGit* (resp. *VersionProjectSVN*) when the VCS is Git (resp. SVN). The precondition named *ExistMM* specifies that the AC named *GeneInterp* requires the existence of the metamodel for its execution.

C. Binding Automation Components to their Implementation

The binding modeler also enables to bind the ACs to their implementation. This is an action (*Action*) that binds an ACs to its implementation. The different kinds of actions (Kermeta action (*KermetaAction*), Java Eclipse action (*JEclipseAction*)) define the technology used to implement the ACs. It is possible to add new kinds of actions (Shell, Groovy,...). An architect writes the implementation of a Java Eclipse (resp. Kermeta) action into an Eclipse plug-in (resp. into its expression (*expr*)). This plug-in registers with an extension point, provided by another plug-in that we have developed. During the registration, the architect declares to which action the plug-in is related (through the id (*id*) of the Java Eclipse Action) and which class of the plug-in is implementing the ACs. This class implements the *run()* method of an interface provided by the plug-in that declares the extension point.

D. Accessing the Execution Context

We have defined a context metamodel in order to store the information about the contexts of use that the ACs need in order to execute. When the metamodel of the process enables to capture this information, then the AC looks for this information into the process model. Otherwise, this is an instance of the context metamodel (*i.e.,* a context model) that stores this information. The context model contains a set of keys that correspond to the information. Each key is associated to a value that corresponds to the value of the information. A process interpreter creates one context model for each process execution and the ACs edit and read this context model. More precisely, when an AC needs an information about the context of use, the person in charge of coding this AC implements according to the process metamodel if the AC looks for this information in the process model or in the context model. During the execution, when an AC looks for an information into the context model and does not find it, then the AC asks for it to the current actor of the process, and stores the information into the context model once it gets the answer.

E. Performing the Process Execution

We have developed a process interpreter that executes a process derived from the SPL. For each process element of the derived process, the process interpreter verifies if there is unresolved variability that is related to this process element. If it is the case, then the process interpreter asks to the current actor of the process to resolve this variability. Once it is done, the process interpreter updates the process under execution according to this variability resolution. Once there is not unresolved variability related to a process element anymore, or if there is not any unresolved variability related to a process element, then the process interpreter executes this process element. If the process element is a work unit, the interpreter launches the execution of the lists of ACs that automate this work unit, according to their order (*i.e.,* initialization, then execution and then finalization). Launching the execution of a list of ACs consists of launching the execution of its ACs, in their order of definition. To launch the execution of an AC associated to a Kermeta action, the interpreter calls a function that dynamically evaluates the expression of this action. To launch the execution of an AC associated to a Java Eclipse action, the interpreter finds the plug-in related to this action and calls the *run()* method of the class implementing the AC. If no list of ACs automates the execution of a work unit, then the interpreter waits for an actor of the project to manually perform the execution of this work unit.

To determine if there is unresolved variability that is related to a process element, the process interpreter looks into the VRM for variation points that reference this process element. If such variation points do not exist, it means that there is not any unresolved variability. If such variation points exist, then the process interpreter verifies if in the RM there are variability specification resolutions that resolve all the variability specifications bound to these variation points. If it is the case, this means that there is not any unresolved variability. Else, this means that there is unresolved variability.

VI. Discussion

In this section we discuss our approach and the tooling support we propose. More precisely, we discuss the limitations of the tooling support we propose, we provide guidelines to implement the ACs and bind them to their contexts of use, and we provide a clarification of the notion of context of use.

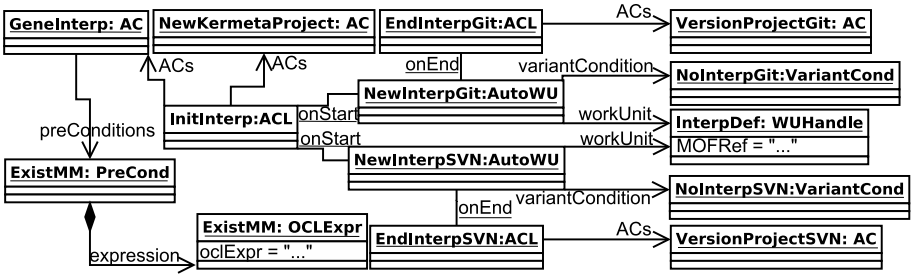

Figure 8: An excerpt of the binding model of the illustrative example

A limitation of the tooling support we propose is that the implementations of the ACs depend on the process metamodel used to define the SPL. Indeed, it is hard written into the implementations of the ACs how they access to the information about their context of use (*i.e.,* by looking into the process model or the context model, according to the kind of information that the process metamodel enables to capture). But if the process metamodel changes and does not enable to capture the same kind of information as the replaced process metamodel, then the way the ACs access to the information about their context of use will need to be updated.

A guideline to implement the ACs is that when the SPL specifies that a tool used to perform a work unit can be substituted by another one, then, in the implementation of the ACs that automate this work unit, the parts that depend on this tool must be decoupled from the parts that do not depend on this tool. Indeed, when a tool used to perform a work unit is replaced, then, in the implementation of an AC automating this work unit, the parts that use this tool must also be replaced. Modularizing the parts that depend on a tool enables to reuse the parts that do not depend on it, when this tool is subject to replacements.

We now clarify the notion of contexts of use of the ACs. The work units, their resources (*i.e.,* tools, roles and work products), their iterations and their sequencing define the contexts of use of the ACs. Indeed, the variants of work units that an AC automates are its contexts of use. And this is the specification of the considered variants of resources, iterations and sequencing related to a work unit that defines which variant of work unit is under consideration. Furthermore, the work units and their resources provide the data (*e.g.,* the URL of a repository) required to execute the ACs.

A guideline to bind the ACs to their contexts of use is that the architect must specify for which sequencing and iteration(s) of a work unit some ACs are relevant, if the ACs depend on these sequencing and iterations. Indeed, the variability of what happens before and/or after a work unit (between the different variants of a process and into one variant) can imply variability of the ACs automating this work unit. Indeed, ACs can become required, useless or unusable, according to what happens before and/or after a work unit. Furthermore, a work unit can vary according to its iterations. For instance, the first iteration of a work unit consists of creating a work, while the following iterations consists of modifying this work. This may imply variations about the ACs that automate this work unit and about the ACs that automate the preceding and following work units. On the other hand, checking the consistency of the pre and post conditions of the ACs and of the work units would ensure that the binding is correct. There is also a need for catching the execution traces in order to know which ACs to execute. Indeed, this enables to know from which path a work unit is entered, by which path a work unit is left, and which iteration of a work unit is occurring.

VII. Related Work

Several approaches exist that enable to reuse processes or that enable to automate the execution of processes. However, to our knowledge, none of these approaches enable to do both things in a satisfying way. In the following we first present the approaches that exist to reuse processes. Then, we present the approaches that automate the execution of processes. We finally present a tool that enables to reuse software processes and to automate their execution, but in a limited way.

Among the approaches that enable to reuse processes, there are approaches [14], [15] that enable to retrieve processes from a set of processes defined in extension (*i.e.,* all the processes are independently defined, without factorizing the common parts between them). Defining processes in extension causes maintenance problems. Indeed, when a part common to several processes evolves, it has to be updated in all the processes it belongs to, which is error prone and time consuming. The process line engineering addresses this problem. Indeed, in a process line the processes are defined in intention, *i.e.*, the parts common to several processes are factorized. In the field of software processes, several approaches rely on Software Process Line Engineering (SPLE) [7] to reuse software processes [2]–[6]. Among them, some approaches target SPEM 2.0 processes [2], [4], [5], while other approaches do not target a specific process metamodel [3], [6]. In the field of business processes, that is closed to the field of software processes, there are also several approaches that rely on process line engineering in order to reuse business processes [16]–[26]. Among them, some approaches target processes that conform to the Event-driven Process Chain (EPC) process metamodel [16]–[18], some approaches target BPMN [27] processes [20], [21], [25], [26], other approaches do not target a specific process metamodel [19], [23], [24], and one approach defines its own process modeling language [22]. However, none of these approaches deals with the automation of the execution of processes derived from a process line, which is not a trivial thing. Indeed, it must be possible to automate the execution of processes whose variability is only partially resolved. Our approach goes further by addressing this problem.

Among the approaches that automate the execution of

processes, one of them focuses on software processes [8]. This approach proposes to automate the execution of SPEM processes by transforming them into WSBPEL [28] processes and by relying on web services to automate tasks. Here web services are similar to ACs. Other approaches focus on business processes and also rely on web services to automate their execution (*e.g.,* [29], [30]). However, these approaches do not deal with the automation of the execution of processes derived from a process line.

Rational Team Concert[6] is a tool that enables to define and reuse software processes and that enables to automate the initialization of a project according to these processes. But the software processes are defined in extension and there is not any facility to select a process according to the requirements of a project. Furthermore, it is not possible to define new components to automate new tasks. Therefore, it is only possible to automate a set of tasks already defined by the tool.

VIII. Conclusion and Perspectives

We propose an approach that enables both the reuse of software processes from an SPL and the automation of their execution. It consists of defining an SPL and of binding the software processes of this SPL to ACs. A software process can be derived from the SPL according to the requirements of a project and the execution of this software process can be automated thanks to its bound ACs. Our approach also enables to automate the execution of processes containing unresolved variability, by enabling to resolve this variability during execution. We provide tools that support our approach. The limitation of this tooling support is that the implementations of the ACs depend on the process metamodel used to define the SPL. We also provide some guidelines to implement the ACs and bind them to their contexts of use.

As perspectives of work, we are applying our approach on industrial projects of a software company, namely Sodifrance[7]. We are also studying the use of CVL to create the binding model that corresponds to a resolved process model, according to the requirements of the projects. This would simplify the binding model because in this model it would only be necessary to specify which work unit of the base process model a list of AC automates, without specifying which variant of the work unit it is. We are also implementing a tool for checking the consistency of the pre and post conditions of the ACs and of the work units. Finally, we are studying if there are other guidelines to implement the ACs.

References

[1] W. S. Humphrey, "The Software Engineering Process: Definition and Scope," *SIGSOFT Softw. Eng. Notes*, vol. 14, no. 4, pp. 82–83, 1988.

[2] T. Martínez-Ruiz, F. García, and M. Piattini, "Towards a SPEM v2.0 Extension to Define Process Lines Variability Mechanisms," in *SERA*, 2008, pp. 115–130.

[3] T. Ternité, "Process Lines: A Product Line Approach Designed for Process Model Development," in *SEAA*, 2009, pp. 173–180.

[4] J. Hurtado Alegría, M. Bastarrica, A. Quispe, and S. Ochoa, "An MDE Approach to Software Process Tailoring," in *ICSSP*, 2011, pp. 43–52.

[5] J. Alegria and M. Bastarrica, "Building Software Process Lines with CASPER," in *ICSSP*, 2012, pp. 170–179.

[6] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel, "Leveraging CVL to Manage Variability in Software Process Lines," in *APSEC*, 2012, pp. 148–157.

[7] H. D. Rombach, "Integrated Software Process and Product Lines," in *ISPW*, 2005, pp. 83–90.

[8] A. Sadovykh and A. Abherve, "MDE Project Execution Support via SPEM Process Enactment," in *MDTPI*, 2009.

[9] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen, "Adding Standardized Variability to Domain Specific Languages," in *SPLC*, 2008, pp. 139–148.

[10] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel, "Improving Reusability in Software Process Lines," in *SEAA*, 2013.

[11] OMG, "Software and Systems Process Engineering Metamodel Specification (SPEM) Version 2," 2008.

[12] ——, "Unified Modeling Language Specifications Version 2," 2005.

[13] ——, "OMG Object Constraint Language (OCL)," 2012.

[14] R. Lu and S. Sadiq, "On the Discovery of Preferred Work Practice Through Business Process Variants," in *ER*, 2007, pp. 165–180.

[15] X. Song and L. J. Osterweil, "Engineering Software Design Processes to Guide Process Execution," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 759–775, 1998.

[16] M. Rosemann and W. M. P. van der Aalst, "A Configurable Reference Modelling Language," *Information Systems*, vol. 32, no. 1, pp. 1–23, 2007.

[17] M. Rosa, M. Dumas, A. H. Hofstede, J. Mendling, and F. Gottschalk, "Beyond Control-Flow: Extending Business Process Configuration to Roles and Objects," in *ER*, 2008, pp. 199–215.

[18] H. A. Reijers, R. S. Mans, and R. A. van der Toorn, "Improved Model Management with Aggregated Business Process Models," *Data Knowledge Engineering*, vol. 168, no. 2, pp. 221–243, 2009.

[19] A. Hallerbach, T. Bauer, and M. Reichert, "Capturing Variability in Business Process Models: the Provop Approach," *Software Maintenance*, vol. 22, no. 67, pp. 519–546, 2010.

[20] V. Kulkarni and S. Barat, "Business Process Families Using Model-Driven Techniques," in *BPM*, 2010, pp. 314–325.

[21] A. Schnieders and F. Puhlmann, "Variability Mechanisms in E-Business Process Families," in *BIS*, 2006, pp. 583–601.

[22] W. Derguech and S. Bhiri, "Reuse-Oriented Business Process Modelling Based on a Hierarchical Structure," in *BPM*, 2010, pp. 301–313.

[23] F. Gottschalk, W. M. van der Aalst, M. H. Jansen-Vullers, and M. La Rosa, "Configurable Workflow Models," *Cooperative Information Systems*, vol. 17, no. 2, pp. 177–221, 2008.

[24] S. Meerkamm, "Configuration of Multi-perspectives Variants," in *BPM*, 2010, pp. 277–288.

[25] E. Santos, J. Castro, and O. Sánchez, J.and Pastor, "A Goal-Oriented Approach for Variability in BPMN," in *WER*, 2010, pp. 17–28.

[26] M. Weidmann, F. Koetter, M. Kintz, D. Schleicher, and R. Mietzner, "Adaptive Business Process Modeling in the Internet of Services (ABIS)," in *ICIW*, 2011, pp. 29–34.

[27] OMG, "Documents Associated with Business Process Model and Notation (BPMN) Version 2.0," http://www.bpmn.org/, 2011.

[28] OASIS, "Web Services Business Process Execution Language Version 2.0," http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[29] G. Gunasinghe and T. Kelly, "Establishing a Standard Business Process Execution Architecture for Integrating Web services," in *ICWS*, 2005, pp. 365–372 vol.1.

[30] X. Guang-cai, W. Zhi-feng, Z. Xin-jia, and J. Guo-jun, "Realization of Business Process Automation Based on Web Services and WS-BPEL," in *ICSSSM*, 2011, pp. 1–5.

[6] http://www-01.ibm.com/software/rational/products/rtc/

[7] http://www.sodifrance.fr/