A Megamodel for Software Process Line Modeling and Evolution

Jocelyn Simmonds, Daniel Perovich, María Cecilia Bastarrica and Luis Silvestre
Computer Science Department, University of Chile
{jsimmond, dperovic, cecilia, lsilvest}@dcc.uchile.cl

Abstract—Companies formalize software processes as a way of organizing development projects. Since there are differences in project contexts, a one-size-fits-all approach does not work well in practice. Some companies use a family of predefined processes, but this approach has a high process maintenance cost. Instead, we define Software Process Lines (SPrL), where a general process with variability is tailored to project contexts. Model-Driven Engineering (MDE) provides a formal framework for defining the models and transformations required for automated SPrl tailoring. However, this approach requires the definition and co-evolution of various types of models and tool support beyond the skills of process engineers, making the industrial adoption challenging. This paper shares our experience using a megamodeling approach to the development of the back-end of our toolset. The megamodel provides a uniform mechanism for process definition, variability, tailoring and evolution, and we hide the MDE complexity through a user-friendly front-end. We report the application of our approach at Mobius, a small Chilean software enterprise.

Index Terms—Megamodel, Software Process Line, Variability

I. INTRODUCTION

Small Software Enterprises (SSEs) define their processes in order to manage their development projects in a systematic way, being able to plan, assign resources, and control the progress of their projects. Having a defined process allows companies to analyze projects’ results in terms of quality and productivity as a basis for improvement. Moreover, if the process is rigorously defined, the company can get an ISO certification or a CMMI evaluation that may give them a commercial advantage. Formally defining the software process allows for using supporting tools for assisted process analysis and management, as well as making evolution easier. Software processes can evolve in different ways, such as, adding or removing activities, changing a work product template, assigning a new role to a particular task, among others.

Software companies get involved in different kinds of projects: new development vs. maintenance, experienced vs. novice development teams, well-known vs. highly innovative technologies, among others. Therefore, the same process cannot address all kinds of projects and still be equally productive in all cases [21]. This can be addressed by defining a series of processes, one for each kind of project, i.e., a process family.

Defining, documenting and following a software process is an expensive endeavor. Defining a whole process family is even more expensive, especially for SSEs. Evolving all these processes independently implies a big effort, as well as a risk of introducing inconsistencies among them. A Software Process Line (SPrL) is a Software Product Line in the process domain, and is one approach to creating process families. SPrL promotes process asset reuse, since common process elements are defined only once, and specific processes are generated through process tailoring [25]. There are different approaches to process tailoring: selecting a process from a predefined set of processes [11], building a software process by configuring a series of process elements [34], or generating particular processes by customizing a general predefined process that includes its potential variability [30], each one with its own advantages and drawbacks.

We follow the third approach, and use Model-driven engineering (MDE) techniques to provide automation. We define processes as models and implement tailoring rules as model transformations that resolve variable process elements, thus consistently generating project-specific processes [18]. The advantage of this approach is that we are sure to obtain the most appropriate process for each project by just defining the project’s characteristics -its context- and executing the tailoring transformation. If the tailoring rules are well-defined, an MDE-based tailoring approach obtains the optimal process. However, it requires defining the software process model that conforms to a particular metamodel, identifying the process variability, the project context model and its corresponding metamodel, as well as programming the tailoring transformation itself. All these activities are very sophisticated and require highly specialized professionals and tools, limiting industrial adoption. Moreover, it is not enough to have a specialized professional defining these modeling artifacts up-front since all of them can evolve: the variable process elements, the context attributes and their potential values, or the rules that define the tailoring transformation.

In [6], we proposed the idea of using a megamodel to address this problem. A megamodel is a model whose elements represent models and transformations [8], and can therefore capture all the elements required by MDE-based SPrL tailoring solutions in a uniform way that ensures integrity by construction. In this paper, we report the experience of formally defining and applying a megamodel for MDE-based SPrLs. The megamodeling approach to tool development has allowed us to achieve the expected functionality, to plan the implementation, and to favor integration. However, it has introduced additional complexity hindering usability, an important quality attribute for tools intended for industrial use. We have addressed this issue by developing a user-friendly front-end
for our toolset [37], with the megamodel as the toolset back-end. In this work, we describe the modeling strategies and decisions made during the construction of our megamodel and tool support. We also show how this megamodel supports process modeling, tailoring and evolution by applying it to the process of Mobius, a small Chilean company. Note that in Chile, almost 90% of software companies are SSEs [33].

In summary, this paper makes the following contributions:
- The definition of a megamodel for MDE-based SPRL modeling and evolution, including a detailed discussion of the trade-offs among different approaches to model processes, project contexts and variability rules.
- The application of this megamodel in an industrial context, discussing our experience with the co-evolution of the different models and transformations.
- A set of lessons learned throughout the two-year process of defining a megamodel in an industrial context, as well as developing supporting tools using megamodeling for the tool back-end hiding its complexity with a user-friendly front-end.

The rest of the paper is structured as follows. In Sec. II we give an overview of MDE-based SPRL and discuss the challenges to industrial adoption. Section III presents our megamodel, including a discussion of modeling strategies and decisions. Section IV describes how this megamodel supports SPRL evolution. Section V presents lessons learned about megamodeling in an industrial context, Sec. VI describes related work in this area, and Sec. VII concludes the paper.

II. SPRL DEFINITION AT MOBIUS

Mobius is a three year old software services company based in Santiago, Chile, that develops integrated software and hardware solutions for Santiago’s public transportation system. Mobius has 20 employees, 8 are directly working in software development and maintenance. Employees perform more than one role in the company, according to the traditional software engineering disciplines (e.g., developer, analyst, tester, etc). Smaller projects typically last a couple of days, while larger development projects are three to four months long.

A. Software Process Modeling

A software process is a structured set of activities required to develop a software system [24], with related artifacts, human and computerized resources, organizational structures and constraints. SPEM 2.0 (Software and Systems Process Engineering Metamodel) [28] is the OMG standard for modeling software processes. It is based on the Meta Object Facility and it is the most popular language used to specify software processes [22].

Mobius’ general software development process (see Fig. 1) is based on the Rational Unified Process (RUP). It is quite detailed in its definition, with 104 tasks, 10 roles and 44 work products, grouped into 4 phases: Initialization, Elaboration, Construction and Transition. Each phase is a collection of activities, where an activity is a “big-step” grouping of role, work product and task uses. Roles perform activity tasks, and work products serve as input/output artifacts for tasks. EPF Composer1 is a popular process modeling tool, which allows the specification of process behavior using UML Activity diagrams. For example, Fig. 2 shows an activity diagram representing the Requirements activity, which is part of Mobius’ general process. Note that the tasks that appear in Fig. 2 correspond to those shown in Fig. 1.

1http://www.eclipse.org/epf
B. Software Process Lines (SPrL)

Mobius started formalizing its development process two years ago, as part of the GEMS project\(^2\). Mobius is one of our partner SSEs in this project. GEMS advocates an MDE-based approach for software process tailoring, creating a Software Process Line (SPrL) [3], which is a software product line (SPL) in the software development process domain [29]. In a SPrL, the process engineer develops and maintains process assets, promoting planned reuse instead of re-actively integrating unanticipated variability in the process model [3]. The process of instantiating a SPrL to a particular project context is called process tailoring [5]. Like the instantiation of a typical SPL, process tailoring is the activity in which software process variation points are resolved in order to adapt a process to the characteristics of a particular project.

Our approach for defining a SPrL at an SSE involves formalizing the general development process and possible project contexts as models, and also implementing tailoring as a model transformation. We currently use the Atlas Transformation Language (ATL)\(^3\) to implement this transformation. This also includes defining the variation points of the general development process, as well as the relationships between context attributes and variation points.

We helped Mobius specify their SPrL, which includes: 1) Organization Software Process, a model of the general development process and its variation points, 2) Organization Projects Context, a model of the possible project contexts at the organization (see Table I), and 3) Variation Decision Rules, a decision model [39] that captures the company’s tailoring know-how. This last model defines how variation points are resolved based on context values. Directly specifying a tailoring transformation in ATL is not an easy task [18], even for an experienced process engineer. To address this issue, we have developed a Higher-Order Transformation (HOT) [40] for automatically generating the process tailoring transformation [37].

Figure 2 shows Mobius’ Requirements activity, which includes two types of variation points: optionality and alternatives. We have marked the variation points in the activity diagrams using annotations as a visual aid, but these are formally defined in the Organization Software Process model. Each variation point in Mobius’ process is associated to a decision rule, specified in the Variation Decision Rules model [37]. Each rule is of the form condition \(\Rightarrow\) conclusion, where a condition is a predicate on the organizational context, and conclusion indicates how a variation point is resolved when the condition is true. For ease of reading, we illustrate these rules along with the variation points in Fig. 2.

Mobius’ model includes 16 rules. For example, for an optional variation point like task Develop vision, Mobius defined the rule (Project type = Non Corrective \(\Rightarrow\) include Develop vision). Figure 2 also includes an alternative variation point, shown as a red decision node. For this variation point, Mobius defined that if Request = Report or Project type = Corrective, then the requirements are defined using a list (Requirements list definition), otherwise they are defined using Use Cases (Use cases definition).

Building the Organization Software Process model required ten sessions, each 4 hours long. Building the Organization Projects Context model and Variation Decision Rules model required five additional sessions. All these activities were carried out up front by the company’s process engineer and the GEMS team.

C. Challenges to Industrial Adoption

The SPrL setup cost has been amortized over several projects, as it now only takes a few seconds to run our tool chain to produce an adapted software process for new projects at Mobius. The tailored software process only includes the tasks and work products that are strictly necessary for the project, and the process configuration step present in RUP-based processes is now carried out in a systematic and replicable manner. However, this approach has two important industrial adoption challenges.

Co-evolution management. There are multiple and complex dependencies between the different models and transformations, and managing these artifacts as part of a Software Process Improvement (SPI) initiative is non-trivial. Given the initial cost of formalizing all the models and transformations, this may hinder the industrial adoption of our approach for other companies. If the SPrL definition is not adequately maintained and evolved, it will erode, making it less useful in time. Evolving the SPrL may be even harder than defining it since relationships and integrity must be correctly preserved. For example, if a new task is added to the Organization Software Process model and it is defined as a variation point, then the process engineer must remember to update the Variation Decision Rules model, specifying how this variation point is resolved during tailoring taking into account the project context potential values. This rule must be consistent with the existing rules, otherwise the tailoring transformation will generate incorrect adapted software processes. Removing elements from a model also requires careful change propagation. If a context attribute is removed from the Organization Projects Context model, then references to this attribute must be removed from the Variation Decision Rules model.

\(^2\)http://dcc.uchile.cl/gems/

\(^3\)http://www.eclipse.org/atl
and the tailoring transformation must be regenerated. If the Variation Decision Rules model is changed (i.e., if a rule is added/removed/changed), then the tailoring transformation must also be regenerated.

**Tool interoperability and usability.** We rely on different types of models, each with its own supporting tool. The advantage is that we use domain-specific model editors, but it also introduces several tool maintenance and evolution issues. We also interact with other tools, we developed injectors and extractors to shield our tool from the evolution of these tools. However, this leads to a proliferation of injectors and extractors. A megamodel helps us plan the evolution of our tool chain. We must also provide a user-friendly front-end for our tool chain, hiding the complexity of our MDE solution from the end user.

Although MDE-based SPrL has many benefits, these challenges difficult its industrial adoption. In the rest of this paper we show how these can be addressed through the definition of a megamodel for SPrLs.

III. MEGAMODELING

In this section we present concepts related and strategies used for formalizing our megamodel.

A. Global Model Management Approach and Notation

Practical applications of MDE are increasingly intensive in modeling artifacts. They involve a large number of heterogeneous and interrelated models which change over time. Megamodelling is a model-based approach for coping with the complexity of managing and evolving such large model repositories [4]. It is centered on the notion of megamodel introduced in [9], which conveys the idea of modeling-in-the-large by establishing and using the global metadata and relationships on the modeling artifacts while ignoring their internal details. Global Model Management (GMM) [27] is a megamodelling approach that offers Eclipse-based tool support for managing modeling artifacts.

The following types of models were considered during the construction of our megamodel. A **terminal model** (stereotype △) is a representation of certain aspects of a system. A terminal model **conforms to** (c2) a **metamodel** (stereotype ▲), and in turn, a **metamodel** conforms to a self-conforming metametamodel that is available in the modeling environment. A **weaving model** (stereotype ↔) is a special kind of terminal model that represents a semantic relationship between elements in different models. GMM relies on the AMW toolset for weaving models, which defines a base and generic metametamodel AMWCore which is extended to define concrete weaving metamodels. An **annotation model** is a special kind of weaving model in which only one model is woven. It is useful to append information to an existing model without changing its metamodel and favoring the separation of concerns.

A **transformation model** (stereotype ➔) is a special kind of terminal model that implements the update or transformation of a set of source models into a set of target models. A transformation model conforms to a metamodel which defines the model transformation language in which the transformation is implemented. This language can be a general-purpose language like Java, or a purpose-specific language like ATL. Currently, our toolset uses both languages. A **transformation engine** executes transformation models. Each execution is captured by a **transformation record** (stereotype ➖) which registers which transformation model was executed, which specific modeling artifacts were used as sources, and which were used as targets. An **injector** is a special kind of transformation model that takes as input a non-modeling artifact and produces a model. Finally, a **megamodel** is a special kind of terminal model that captures the semantic relationships among a set of modeling artifacts.

We further characterize the modeling artifacts according to their provenance or scope, using different colors to outline artifacts. Standard MDE artifacts, like the AMWCore, are outlined in red. Organization-independent artifacts developed by our team are outlined in green, while organization-specific artifacts created by our team and participating SSEs are outlined in blue. Finally, project-specific artifacts are outlined in black.

B. SPrL Megamodel

In this section we present our megamodel for formalizing and evolving a SPrL. An extended version of this discussion can be found in [38].

1) Modeling Processes: We identify three strategies for modeling the Organization Software Process: using an ad hoc (M1), standard (M2) or controlled (M3) modeling language. Chilean SSEs usually favor M1, using text documents or wikis to capture best practices, guidance, and frequent workflows. The advantage of M1 is that SSEs have an explicit representation of the shared process knowledge, but the lack of formalism makes it hard to analyze and maintain these process descriptions.

Picking a well-defined or documented modeling language with existing tool support makes strategy M2 attractive. For example, EPF Composer produces models conforming to the Unified Method Architecture (UMA) [15] metamodel, which sought to improve the SPEM 1.0 standard (EPF does not yet support SPEM 2.04). Note however that neither UMA nor SPEM 2.0 directly support the specification of the fine-grained behavior of a process. In particular, EPF Composer uses UML Activity diagrams to define process workflows, which are stored in a separate model. Moreover, different customer organizations may rely on different modeling languages. In our experience with Chilean SSEs, some use EPF Composer, but others use Enterprise Architect. Here, UMA is the common modeling language, but it is important to shield our toolset from this kind of tool diversity.

Strategy M3 takes care of this variability by introducing a controlled modeling language; we have called it eSPEM [18] (experimental SPEM). This modeling language includes the subset of UMA and SPEM constructs needed for the specification, enactment and improvement of SPrLs for SSEs. As

4http://www.eclipse.org/epf/composer_architecture/
we control the evolution of this language, we also control the evolution of the supporting tools. Note however that in our case, the goal is not to define yet another modeling language, but rather to allow SSEs to continue using the tools they currently use. Thus, we need a mechanism for importing processes modeled in a standard language into processes modeled in our controlled language. To this end, we developed an Injector from UMA into eSPEM, and an analogous one can be implemented to import SPEM 2.0 processes. Figure 3 illustrates the modeling artifacts involved in strategy M3, specific to the UMA example.

2) Modeling Process Variability: We need to decide how to represent process variability, as well as the mechanism for resolving variability during SPrL configuration, so as to enable automated process tailoring. In principle, any process element can vary. Here, we restrict ourselves to task variability (optional and alternative tasks), as these are the constructs used by our partner SSEs, but our approach can be extended to handle other types of variability.

Process variability can be captured using existing UMA constructs. Tasks have a dual representation in UMA: the TaskDescriptor represents a reusable specification of a task, and the Task metaclass represents a concrete task instance appearing in a process work breakdown. Linking a TaskDescriptor to a Task specifies what task must be carried out. Unlinked TaskDescriptors must be self-contained, specifying the task to be performed.

Optionality can only be defined in TaskDescriptor. The BreakdownElement (super metaclass of TaskDescriptor) defines an IsOptional attribute, used to indicate if the breakdown element is optional during process enactment. Optionality cannot be stated in a Task since more than one TaskDescriptor may be linked to a same Task, and they may not all have the same IsOptional value. Alternatives, however, cannot be modeled at the TaskDescriptor level because UMA’s variability resolution mechanism is modeled in the VariabilityElement metaclass, which is only super metaclass to the Task metaclass, and not the TaskDescriptor metaclass. In particular, we use the replaces variability type to state that a task replaces another. Role and work product variability can also be modeled like this.

Figure 4 shows a concrete example, where we use a UML 2.4 Object diagram to present an Organization Software Process in UMA. The example’s method library includes four task definitions: t1, t2, t2a and t2b, where t2a and t2b are alternatives to t2. The example also includes a delivery process consisting of two sequential task instances: td1 followed by td2. Task instance td1 is optional and linked to task definition t1, whereas td2 is linked to t2 and is mandatory. This Organization Software Process model has four valid configurations, depending on how process variability is resolved: 1) t1, t2a, 2) t1, t2b, 3) t2a, and 4) t2b.

Note that this section only explains how variable process elements are modeled, without specifying how variability is resolved. This information is captured using variation rules in the Variation Decision Rules model, discussed in the next section.

3) Modeling Variation Decision Rules: Variation rules capture the context conditions under which the different process variation points are resolved during tailoring. We identify three strategies for capturing these rules: using an existing process modeling language (V1), specializing an existing process modeling language (V2), and capturing rules using a separate artifact (V3). Strategy V1 uses the constructs that are already available in the process modeling language to capture rules in order to preserve tool support. This strategy has several drawbacks: the language chosen may not include the constructs needed for rule resolution, nor those needed for characterizing development scenarios. Also, it ties SPrL definition to a particular modeling language.

Implementing strategy V2 requires extending an existing language. For example, we can define our own UMA extension that refines the Task and TaskDescriptor metaclasses with constructs for specifying rules, so that these can be attached
to tasks. We call this extension xUMA to simplify the presentation. Rule conditions can be included by further extending xUMA (flexible, but harder to maintain), or using a language like OCL (active community, but harder to learn). These conditions refer to context attributes, which are external to the process itself, so additional xUMA extensions are needed. The main advantage of this strategy is that it provides domain-specific constructs for capturing variation rules. Unfortunately, there is no separation of concerns: everything is captured in a single terminal model. We also lose domain-specific tool support, e.g., EPF Composer does not allow rule specification, so a UMA model would have to be transformed into xUMA, and only then can conditions be added.

Strategy V3 maintains the separation between process modeling and variability rules, which simplifies tool implementation. Separate terminal models are used to capture the SPtL (process - Organization Software Process, context - Organization Projects Context and variation rules - Variation Decision Rules). The first two terminal models are self-contained, while the Variation Decision Rules model references elements from the other two models by name, meaning that an element may appear in more than one terminal model. This aspect of strategy V3 can be improved by making Variation Decision Rules a weaving model. In this case, we would need a customized metamodel for weaving models, as the simple links provided by AMWCore\(^2\) are not expressive enough for capturing decision rules. We can improve strategy V3 by making it independent of the process modeling language used (replacing UMA with eSPEM, as discussed in Sec. III-B1).

Our toolset currently implements strategy V3 using implicit links. These links are enforced through the use of user-friendly tools to populate the Organization Projects Context and Variation Decision Rules terminal models. An ad hoc language is used to specify rules, which supports equality, conjunction, disjunction, and precedence. We made this decision because of time constraints, but we plan to update our toolset to use a weaving model and OCL in the near future.

4) Context Modeling: The Organization Projects Context terminal model captures the different contexts where the Organization Software Process can be enacted. A specific configuration of the Organization Projects Context model, namely a Project Context, is created by selecting a single value for each context attribute, for all context dimensions. The project manager must generate a context configuration for a project before it begins, as this context characterization is used to resolve process variability and generate a process specifically tailored to the project’s context. While the Organization Projects Context terminal model is specific to the organization, the Project Context terminal model is specific to a single project at an organization. We have identified five modeling strategies for modeling project contexts: a shared metamodel (C1), separate metamodels (C2), a metamodel extension (C3), a weaving model (C4), and an annotation model (C5).

Strategy C1 is possible because a Project Context is a configuration of the Organization Projects Context. While the latter defines the set of possible values for each Attribute of each Dimension, the former is just the selection of a single Value for each Attribute. This means that we can use a single metamodel to express both terminal models. This strategy is easy to implement but the resulting models must be validated for well-formedness using an external tool, since the metamodel provides constructs that should not be used in both types of models. Strategy C2 is similar to C1, but defines a separate metamodel for Project Context. Here, model well-formedness is achieved at the expense of simplicity.

Strategy C3 is similar to strategy C2, but here the Project Context metamodel is an extension of the Organization Projects Context metamodel, adding a construct for selecting Attribute Values. This solution fixes the problem of modeling element replication between models, but reintroduces the problem of invalid references. For example, a Project Context should not include a set of Values for an Attribute, but it can with this approach, like C2. This can be fixed by defining a base Context metamodel, which only includes the constructs common to both terminal models, as well as two extensions of this base model, one for each type of terminal model. This ensures that all terminal models will be well-formed, and eliminates modeling construct duplication as common elements are defined in the base Context metamodel. However, this strategy introduces another modeling artifact that must be maintained.

Strategy C4 makes the relationships between the two terminal models explicit, by defining a weaving model. In this case, the elements Dimensions and Attributes can be removed from the Project Context metamodel, leaving only Selection, which is linked to the corresponding Values in the Organization Projects Context. No constructs are duplicated in this approach, but the Project Context metamodel consists of just one construct: Selection. As such, this strategy overcomplicates the megamodel, without real justification.

Finally, strategy C5 tries to solve the problem introduced by strategy C4, while preserving the advantages of the previous strategies. A Project Context model only adds information to an existing Organization Projects Context model. In other words, it is an annotation model, a particular kind of weaving model that has a single woven model. This strategy is shown in Fig. 5, where the Project Context weaving metamodel

\(^2\)http://www.eclipse.org/gmt/amw
only defines a type of link that represents Value selection. This strategy does not have any of the problems the previous strategies had.

Our toolset currently follows strategy C1 for its simplicity, where a user-friendly front-end is used to build both terminal models and keep them consistent. In practice, the process engineer at a SSE uses our toolset to define a single Organization Projects Context terminal model for the organization, and at the beginning of each project, the project manager uses our toolset to define the specific Project Context terminal model. As future work, we plan to fully embrace strategy C5 and make our toolset rely on standard MDE constructs, so that well-formedness is enforced at the MDE level.

5) SPRL Tailoring: We can now formally define SPRL process tailoring using the megamodel. In our approach, process tailoring is divided into two steps: i) generation and ii) application of the tailoring transformation. Figures 6 and 7 show how these steps are formalized using our megamodel, respectively.

i) Once the Organization Software Process, Organization Projects Context and Variation Decision Rules terminal models have been defined by the process engineer, we can generate the tailoring transformation. This is done using an organization-independent Higher Order Transformation (HOT) [40] that takes these three models as input and automatically generates an organization-specific process tailoring transformation. In Fig. 6, Tailor Generator and Tailor represent the HOT transformation and the tailoring transformation, respectively. As reported in [36], the HOT has been implemented in Java, and the tailoring transformation conforms to ATL. A Tailor Generation Record is registered each time the HOT transformation is executed.

ii) Before starting a new project, the project manager must define the project’s context values (Project Context terminal model in Fig. 7). Table II shows two examples of project contexts for Mobius: New Development project (A), and Corrective Maintenance project (B). For brevity, only the attributes included in the rules in Fig. 2 are shown. The tailoring transformation Tailor generated by the process engineer takes as input one of these contexts, as well as the Organization Software Process, generating the Adapted Software Process for that project context. For example, if the project manager chooses Context A, the process shown in Fig. 8a is generated for the new project. Choosing Context B results in the process shown in Fig. 8b, which is quite different. Each application of the tailoring transformation is registered (see Tailoring Record in Fig. 7).

IV. SPRL EVOLUTION

We now discuss how our megamodel enables process line evolution, using changes requested by Mobius.

A. Unit of Evolution

Change can occur at any level of our megamodel: both in the organization-independent and -specific models, as well as in the project-specific models. Moreover, the standards used by the GEMS project can also change (SPEM 2.0, UMA, ATL, etc.). Not being able to deal with these types of changes reduces the expected lifetime of an SPRL, and not dealing with...
them in a uniform and systematic manner reduces the chances of industrial adoption of our approach.

Also, we expect that some types of changes will be more frequent than others. For example, the organizational process model, the organizational context, and tailoring rules may change soon after SPrL definition, to account for differences between the modeled process and the process actually carried out by the company. These artifacts may also change as a result of an SPI process. Other elements, like the HOT or the eSPEM model, are less likely to change, as they have stabilized over the course of the GEMS project.

Thus, our SPrL is under continuous evolution, as defined in [16]. In practice, it requires a higher maintenance effort than small and simple systems. Supporting tool availability play a gravitant role allowing evolution to be an easier task, avoiding erosion [31][19].

This is one of the main advantages of our megamodeling-based SPrL approach: as dependencies between the different models and transformations are explicit in the megamodel, our tools can offer automated change impact analysis without requiring additional effort than that already required to define the SPrL. By reducing the SPrL maintenance effort, we expect to see an increase in the lifetime and adoption of the SPrLs defined by our partner SSEs.

B. Organizational Process Management

The definition of a software process using EPF Composer is labor-intensive, even for a small process. So taking advantage of this investment is appealing. Changes such as adding a new template for a work product, or changing the role in charge of a task may be quite frequent. Also, process variation points may change over time. For example, if the template assigned to a work product needs to be changed, it should be done directly in the EPF Composer and then the Injector needs to be re-run. The updated Organization Software Process will refer to the new template for all future projects.

For example, Mobius wants to update their Organization Software Process, removing the System architecture definition task from the Requirements activity shown in Fig. 2. In order to evolve the SPrL, the process engineer must first update the Organization Software Process using EPF Composer. This will leave the Variation Decision Rules in an inconsistent state, as it now refers to a process element that no longer exists. The toolset notifies the process engineer when there is an inconsistency, asking her/him to fix it. This in turn triggers the execution of the HOT, producing a new tailoring transformation (see Sec. III-B5). Now, when the process manager picks Context A for a new project (see Table II), the resulting adapted process is similar to the one shown in Fig. 8a, but it does not include the System architecture definition task because it is no longer present in the Organization Software Process.

C. Context Evolution

Context values vary for each project, so a new Project Context should be created for each one. Afterwards, the tailoring transformation is executed to obtain the Adapted Software Process. This type of change is not too drastic and it happens for every new project, so tool support is essential. For example, if the project manager picks Context A, but then changes the value of the context attribute Usability to Low, the resulting adapted process is similar to the one shown in Fig. 8a, but it does not include the UI design task.

On the other hand, the Organization Projects Context may require more drastic changes, such as adding, modifying or deleting context attributes and/or their values. If attributes or values are removed or modified in the Organization Projects Context, then the Variation Decision Rules must be checked for consistency. Adding, updating or removing attributes or values may require adding new rules, updating existing rules, or removing existing rules.

Either way, the Tailor Generator must be re-run if the Variation Decision Rules changes, creating a new version of the tailoring transformation. For example, if the context attribute Request is removed from the Organization Projects Context, the variation decision rule attached to the decision node in Fig. 2 must be updated to Project = corrective. Re-running the HOT will create a new version of the tailoring transformation. Now, when the project manager picks Context A, the resulting adapted process is exactly the same as the one shown in Fig. 8a, since Project type = corrective is true in this context.

D. Tailoring Rule Evolution

Tailoring rules should evolve when the Adapted Software Process generated by the tailoring transformation is not as expected. If the problem is in the Organization Software Process, then the procedure described in Sec. IV-B must be followed; if it is in the Project Context, then the procedure in Sec. IV-C must be followed. However, if both models are correct, it means that the relationship between the two is what is incorrectly defined, and therefore the Variation Decision Rules must be analyzed, checking if the mapping between the context attributes and variation points is correctly specified. Once this model is updated, the HOT must be re-run to create a new tailoring transformation.

V. Lessons Learned

Mobius’ process engineer validated that for all project contexts he defined, the resulting adapted processes were as expected. Similar results were obtained when changes were made to the software process and the tailoring rules. However, he pointed out that he doubted that during their day-to-day activities, the Mobius team would be willing to use this toolset as is. This is due to their lack of familiarity with concepts like variability, process tailoring and tailoring rules.

As such, we detected several factors that affect the adoption of software processes in industry: (a) the complexity, expressiveness and understandability of the notations and languages, (b) the perceived cost-benefit relation of coping with variability and of successfully achieving variability resolution for particular development scenarios, (c) the degradation of
the captured process model with respect to the actual process enacted due to an ill-managed change and evolution, (d) the availability and usability of tool support, and (e) lack of SPI and MDE skills in SSE employees.

While MDE provides a solid foundation for our approach and toolset, a naïve application of MDE relying only on modeling-in-the-small constructs rapidly degenerates in a large and complex set of modeling artifacts, hindering co-evolution. On the other hand, an application of MDE relying on modeling-in-the-large constructs promotes control. The various modeling artifacts involved in the megamodel are well-classified and cataloged, and their interrelations are made explicit. Thus, the impact of change can be readily identified and visualized. For instance, whenever a change is required in a particular modeling artifact, the megamodel can be navigated to analyze which related modeling artifacts should be preserved, re-examined, or re-generated.

We did not always adopt the best MDE practices when creating our modeling artifacts and megamodel. The first reason is that there are few reported applications of sophisticated MDE techniques like megamodels and HOTs. Another reason is that the megamodel emerged from existing modeling artifacts generated by the GEMS project. Finally, tool support for the various MDE constructs is at varying levels of maturity, so in various cases we picked a simpler modeling strategy over a more pure MDE strategy. We also had to implement several user-friendly front-ends to hide the complexity of using MDE tools from our industrial partners.

The megamodel provides the big picture of all involved artifacts and the role they play in the overall solution. This helped us organize the GEMS project, and we can now improve our toolset in a focalized manner to improve industrial adoption, by mitigating risks, addressing problems, or assisting in their resolution. For instance, the choice to introduce the proprietary process modeling language eSPEM was to shield our solution from changes to the process modeling approach used by SSEs. While it is not frequent for a SSE to change its process modeling language, this scenario is recurrent since we interact with multiple SSEs. We also introduced a tailoring transformation generator. While our toolset initially relied on a custom-built tailoring transformation, this approach did not scale as we worked with more SSEs, so we abstracted the variation decision rules in a purpose-specific model instead of embedding them in the tailoring transformation. This change significantly improved the usability of our toolset.

VI. RELATED WORK
Process Variability Modeling. SPEM 2.0 defines four primitives for specifying variability between two process elements: contributes, replaces, extends, extends-replaces, but it is hard to predict how variability relations interact with each other, since instances of these relations may override each other, and these are rarely used in practice [26]. In that same article, Martínez et al. propose vSPEM, a SPEM extension that allows the direct specification of process variability. This notation is intuitive, but tool support is an academic prototype that is no longer supported, making it a poor candidate for industrial adoption. Feature Models [20] (FM) can also be used to model process variability. This approach has been explored in [18], [13], but it is limited in that process lines have few variation points and many common elements, so the resulting FMs are hard to maintain.

Business Process Models (BPM) [41] are used to model general business processes. There have been several proposals for capturing variability in BPMs. Hallerbach et al. [14] introduce the notion of “options”, sequences of change operations over a base BPM; variant BPMs are then created by executing one or more options. The drawback of this approach is that variability is specified operationally, so option interaction makes evolution difficult. Configurable Event-Driven Process Chains [23] (C-EPC) is another approach for modeling business process variability. C-EPCs are directed graphs, where nodes are annotated with constraints to indicate if they are mandatory or optional, and variant C-EPCs are created by processing these annotations. There is no separation of concerns, so these models become harder to evolve.

Variability Evolution. Some empirical studies [10][42] argue that existing approaches offer weak tool support for product line evolution. Without adequate tool support for change propagation, it is hard for engineers to understand the possible impact of their changes to feature hierarchies, and as a result, evolution is primarily limited to adding features and decisions [7]. As a consequence, the models supporting SPL definition and derivation are seldom refactored, leading to an increase in model complexity over time. Several ad hoc traceability models have been proposed as a way to control product line evolution [1][35][2]. Model-based approaches for modeling and evolving product lines have also been defined in the literature [17][12][32]; however, to our knowledge, we are the first to generalize this approach by defining a megamodel for process lines, enabling process line evolution in a controlled and uniform manner.

VII. CONCLUSION
Megamodeling is a feasible solution for the definition and evolution of software process lines. Megamodeling allows managing the complexity of approaches that are intensive in modeling artifacts, and it is even more relevant when those modeling artifacts evolve. Using a megamodel in an industrial context allows the classification and characterization of all the modeling artifacts and their relationships, and it also serves as a back-end for centralized and integrated tool support. This back-end is sufficiently generic in that we can use it to set up SPRPs at different SSEs, and participating SSEs now have a SPRl that can be more easily analyzed and improved.

Ongoing work. Further integration of tool support is required by our industrial partners. As EPF Composer is a standalone product, process modeling is performed in a non-MDE environment. Provided that its underlying platform is Eclipse, we are in the process of adding the functionality provided by this tool directly into our toolset, thus providing a single-IDE experience to process engineers and project managers.
ACKNOWLEDGMENT

This work has been partly funded by Project Fondef GEMS IT13I20010 and Luis Silvestre was also supported by PhD Scholarship Program of Conicyt, Chile (CONICYT-PCHA/2013-63130130).

REFERENCES


