A Family-based Framework for i-DSML Adaptation

Samson Pierre, Eric Cariou, Olivier Le Goaer, and Franck Barbier

Université de Pau / LIUPPA, PauWare Research Group, BP 1155,
F-64013 PAU CEDEX, France
{firstname.name}@univ-pau.fr, http://www.pauware.com

Abstract. One of the main goals of Model-Driven Engineering (MDE) is the manipulation of models as software artifacts. Model execution is in particular a means to substitute models for code. Precisely, if models of a dedicated Domain-Specific Modeling Language (DSML) are interpreted through an execution engine, then this DSML is called interpreted-DSML (i-DSML for short). The possibility of extending i-DSML to adapt models directly during their execution, allows the building of adaptable i-DSML. In this article, we demonstrate that specializing adaptable i-DSML leads to the potential definition of accurate adaptation policies. Domain-specificities are the key factors to identify adaptations that really make sense. In effect, we introduce the concept of family as a mean to encapsulate adaptation operations that are attached to a particular domain. Families can be specialized with the special purpose of defining a hierarchy of adaptation contexts.

Keywords: model execution, adaptation, i-DSML, models at runtime

1 Introduction

The main goal of Model-Driven Engineering (MDE) is to cope with productive models to build software. This can be commonly achieved by generating the code of the software from the models. On another hand, it is also possible to directly execute a model. In this case, the software system is an execution engine implementing an execution semantics and interpreting a model. Such a model is written in an interpreted Domain-Specific Modeling Language or i-DSML for short [8]. With i-DSML, the ability to run a model prior to its implementation is a time-saving and henceforth cost-saving approach for at least two reasons: (a) it becomes possible to detect and fix problems in the early stages of the software development cycle and (b) ultimately the implementation stage may be skipped. One slogan associated to i-DSML could be "*what you model is what you get*" (WYMIWYG).

Meanwhile, software adaptation and self-adaptive software [15] have gained more and more interest. The runtime adaptation problem is commonly tackled as a two-stage adaptation loop (analyze–modify). In the MDE field, one of the most prominent way to implement this loop is *models@run.time* [2], where models are

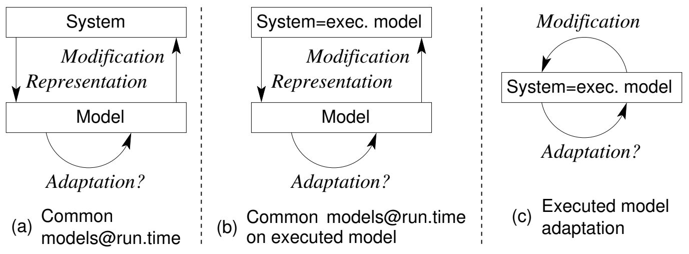

Fig. 1. Adaptation loops

embedded within the system during its execution and acting primarily as a reasoning support (case (a) in Fig. 1). The model is representing the current state of the system and the necessity of adaptation is checked through it. The required adaptation actions are then processed on the system. For adapting a model execution, these *models@run.time* principles can of course be applied (case (b) in Fig. 1). One can notice that in this particular case, there are two models at runtime. The first one is the executed model and the second one is the model representing its state in an adaptation purpose. As the content of the latter is based on the content of the former, this introduces a kind of redundancy between the two models which are hence containing similar or derived elements. In this case, why not directly integrating elements dedicated to the adaptation in the executed model? Even if it leads to complexify the model, it avoids the main disadvantage of *models@run.time* which is to maintain a consistent and causal connection between the system and the model for the model being a valid representation of the system at runtime. Now (case (c) in Fig. 1), the model is directly self-interrogating for managing its own adaptation. Such adaptable and executable models are written in an adaptable i-DSML [5, 6].

In this paper, we focus on the direct adaptation of an executed model (case (c) in Fig. 1) with the definition of adaptable i-DSML. To that extent, we propose an example about a homemade process modeling language. Through this example, we show that specializing the i-DSML leads to enabling automatic and relevant adaptation policies. Indeed, with general-purpose models (and without strong link to any particular business content), it is often difficult, even impossible, to define automatic adaptation actions. Adding new elements on the metamodel or restricting the space of possible models through additional constraints can unlock this situation. The concept of adaptation family is proposed for managing adaptable i-DSML specialization. A family is composed of a specialized metamodel and associated adaptation policies. Families may inherit from each other allowing the definition of hierarchies of families. Inheritance naturally offers the reuse, factorization and specialization of adaptation policies as for code in object-oriented programming.

The rest of this paper is organized as follows. The next section presents an i-DSML defining timed processes and shows that, in case of delay in the process execution, no adaptation action can be established. Sect. 3 defines the concept of families for managing adaptation. Sect. 4 presents some families for the i-DSML of timed processes and concrete adaptation policies. Finally, related work is discussed before concluding.

2 i-DSML Adaptation: a Working Example

Let us consider the i-DSML named *Process Description Language* (*PDL*), that is intended to model any kind of processes as an ordered list of activities. It is freely inspired from standard process languages like SPEM [13] or BPMN [1], which are typically coupled with workflow engines for their execution. Here, this is a simple version that supports parallel activities and includes time concerns. The metamodel and the execution semantics of such an i-DSML are described prior to an illustration of the latter in action is provided. Then, questions about its possible adaptations are raised.

2.1 Definition of the *PDL* metamodel

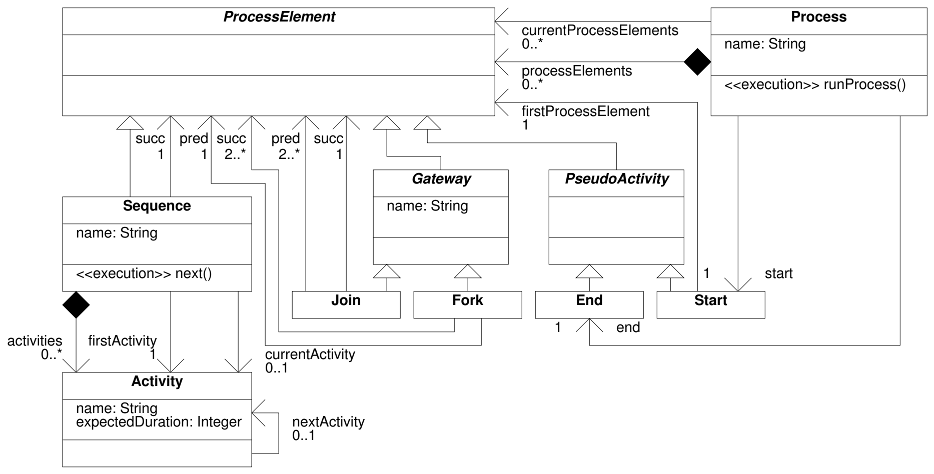

Fig. 2. Definition of the *PDL* i-DSML

Model execution and i-DSML have been widely studied, for instance in [3, 4, 6, 8, 9, 12]. All these works establish a consensus on model execution. Accordingly, in this section, the *PDL* i-DSML is described following the characterization of [6]. Fig. 2 defines its metamodel. A metamodel of an i-DSML first contains a static part. This part is simply a metamodel commonly defined for design purposes. Here, the goal of this static part is to define the elements which aim at forming the structure of a process. These elements are manifold and abstracted through the `ProcessElement` meta-element. The main concrete process element is a sequence containing a set of activities. Activities within a sequence are ordered as each activity (except the last activity) has a next one. Each activity has an expected duration, which is the ideal time lapsing to complete the task. Sequence of activities can be parallelized through gateways. A fork aims at making several sequences parallel whereas a join is a synchronization point of several sequences. Finally, each process contains two pseudo-activities defining the beginning and the ending of the process.

Additionally, an i-DSML metamodel contains a dynamic part. Its goal is to be able to specify the current state of the model during its execution. Here, the dynamic part consists of two meta-associations in Fig. 2. The first one expresses for a process which process elements are currently active. The second one refers, for a sequence, to its current active activity, if any. The combination of the two gives the global state of the model under execution which is modified after each execution step.

The static and dynamic parts of the metamodel are augmented with OCL invariants expressing the well-formedness rules, for instance, there is no cycle between activities. Due to lack of place, they are not presented.

Finally, the metamodel of an i-DSML is associated with an execution semantics. Its goal is to express how the elements of the model are evolving during the execution. Concretely, the execution semantics only modifies the dynamic elements of the model and is implemented through a set of execution operations that can be attached to meta-elements. Here, the execution semantics is embedded within the `runProcess()` operation of `Process` and the `next()` operation of `Sequence`. For the sake of clarity, these special operations are prefixed by an `<<execution>>` conceptual stereotype. The `runProcess()` operation launches the process execution. Its first action consists in executing the start element of the process. Then, after the end of its execution, it executes its successor elements and so on, until reaching the last element of the process. Executing a sequence consists in executing its `next()` operation. If the sequence is just launched, it executes its first activity. Otherwise, it executes the next activity of the current one. Once an activity is finished, the `next()` operation is recalled and so on until reaching the end of the sequence. Executing a fork consists in executing each of its successor sequences. Executing a join consists in waiting for each of its predecessor sequences to be finished before executing its successor one.

2.2 A Software Development Process Defined Using *PDL*

As a familiar example, we choose to model a typical software development process (Fig. 3, top part). This process contains four sequences (represented as dashed rectangles): specification (`Specify`), implementation (`Implement`), documentation (`Document`) and distribution of the software (`Distribute`). The specification is the first task of the process whereas the distribution is the last one once everything else is finished. Between the two, the implementation and the documentation are realized in parallel. This is achieved through the fork named `FID`

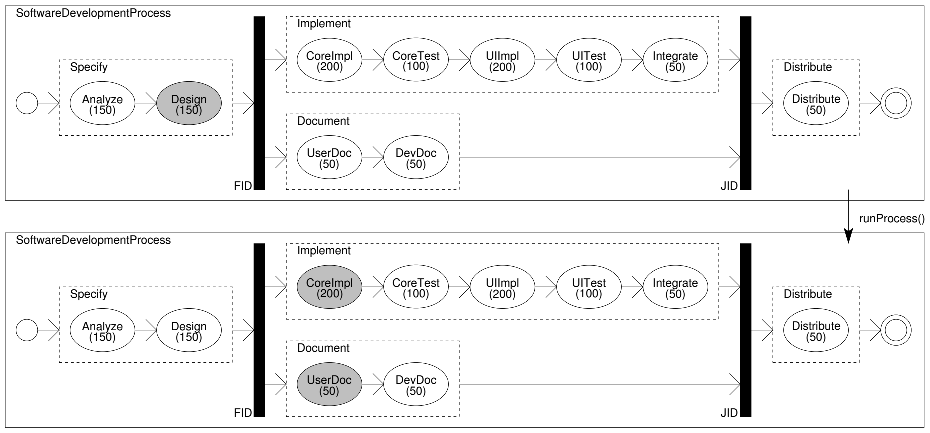

Fig. 3. A model conforming to the *PDL* i-DSML and its execution

(*Fork for Implement and Document*) and the join `JID` (*Join for Implement and Document*). The specification contains two activities (represented by ellipses): analysis followed by design. The number beneath the activity name, here 150 for both, is the expected duration of the activity. The documentation contains user documentation and developer documentation activities. The implementation is the longest sequence: it begins with the core implementation, that is business logic and services implementation (`CoreImpl`) and its associated test activity (`CoreTest`). Next, the user interface is implemented (`UIImpl`) and tested (`UITest`). Finally, all implementations are integrated.

When the execution engine takes the model as input, the execution trace is formed by a collection of snapshots that correspond to every state of the model after an execution step. We start by the first activity of the first sequence (the `Analyze` activity). Then we continue with the next activity of this sequence (the `Design` activity). This situation is represented at the top of Fig. 3 where the current activity is filled with the gray color. After this, we encounter the fork, so, the first activity of each successor is activated (the `CoreImpl` and the `UserDoc` activities). This situation corresponds to the bottom of Fig. 3. Once each activity of these two sequences has been executed, we are able to cross the join. The last activity executed is then `Distribute`.

2.3 Toward the Adaptation of a Process

Still following the characterization of [6], an i-DSML is extended following two ways for becoming adaptable. First, the metamodel is extended with elements dedicated to the adaptation and second, an adaptation semantics is associated with the metamodel and implemented by the execution engine so that it is turned into an adaptation engine. The adaptation semantics aims at acting on the model for adapting it. This adaptation may have impacts on the whole model, including the modification, creation or deletion of static, dynamic or adaptation elements.

Concretely, the adaptation semantics is implemented through two kinds of operations. The first kind is query operations returning a boolean and expressing if an adaptation is required or not. They are called adaptation checks. The second kind is adaptation actions that apply the adaptation on the model.

Concerning the *PDL* i-DSML, at first glance, a situation requiring adaptation is when the process has been delayed. However, this necessitates to evaluating the laps between the expected duration and the real elapsed time. So, an adaptation element has to be added on the metamodel: an `elapsedTime` attribute in the `Process` meta-element. This basic extension of the metamodel is intuitively evident since having an elapsed time is a logical complement of an expected duration for an effective process execution. It is now easier to implement a check operation that determines if we are late during the process execution. It concretely requires comparing the expected durations of all already finished activities to the real elapsed time.

Now the question is: “If we are late, what must we do?”. The answer is: “In the current definition of the i-DSML, we do not know!”. Indeed, there is no obvious adaptation action that can be processed without additional information. We may imagine removing unnecessary activities in the rest of the process. Which ones? An arbitrary erasure is clearly not a good idea. In the example, the documentation and testing activities may be bypassed. We know that because we have business knowledge on, in general, what a software development process really is. Documentation is never at the core production of the subject software and testing, in the worst case, can possibly be skipped if really required. The problem is that the execution engine has no business vision and then, has not this business knowledge. The engine agnostically processes the adaptation for any kind of model, being a software development process or a cooking recipe. Another idea would be to parallelize some activities to reduce the time of the process execution. Typically, from a business knowledge viewpoint, we know that the development of the business part can be potentially done in parallel with the user interface development. Again, the engine does not have this business knowledge.

As a conclusion, with the basic definition of the i-DSML (including a small and evident extension), it is not possible to know how to adapt a *PDL* model. In other words, it is not possible to define an adaptation semantics. The model is too general. However, we foresee that with additional information, adaptation actions can be defined and make sense. The metamodel then needs to be specialized to embed this additional information. More generally, the most specialized and constrained a metamodel is, the most automatic manipulation of the model is possible. Besides, the most defining relevant adaptation semantics is made possible. In the next section, we define the concept of family which is an i-DSML specialization associated with a dedicated adaptation semantics. Then, in Sect. 4, we define concrete adaptation families for the *PDL* i-DSML.

3 Family-based Framework for i-DSML Adaptation

Assuming the fact that from a minimal i-DSML one can create various extensions, each providing a foundation for dedicated adaptations, it becomes highly desirable to organize all these software pieces. That is why we propose in this paper the adaptation families and the specialization relationship between them.

3.1 Definition of a Family

Each metamodel of an i-DSML, with the definition of its meta-elements, leads to define a set of operations that make sense and are implementable based on these meta-elements and their associated constraints. These operations are those defining an execution or an adaptation semantics. Then, as these operations are tied with a given metamodel, we propose to logically group them under the form of a family. Here is the definition of a family:

Definition 1. *A family brings together a metamodel and a set of associated operations (execution operations, adaptation checks and adaptation actions). It provides guidance to a software designer that can glue together operations available in this context.*

Hence, a family is like a frame in which an engineer can dip into extant elements of solutions with confidence. Afterwards, she/he is responsible for their correct orchestration. A family is identified by an unique name referencing its metamodel and contains three kinds of elements:

1. Execution operations: operations that control the execution flow of the model.
2. Adaptation checks: boolean operations expressing if the current model is aligned with the execution environment or is respecting specific constraints. If not, adaptation must be undertaken.
3. Adaptation actions: operations that modify the content of the model in an adaptation purpose.

We call "attributes" all these elements within a family.

3.2 Family Specialization

The conclusion of the discussion of Sect. 2.3 was that specializing a metamodel is relevant for defining adaptation: the *PDL* i-DSML has first been extended and then, the discussion concluded on the necessity of extending it one step further to have the ability to define concrete adaptation semantics. As a consequence, we propose the specialization of families and thereby to build a hierarchy of families for defining adaptation semantics.

Specializing a family is based on specializing metamodels. Model typing and subtyping, that is specialization relationships between metamodels or metamodel parts, have been defined in [11, 18, 19]. The metamodel specialization we use in this paper is based on their definition. A metamodel defines a structure (a set of associated meta-elements) and is augmented with a set of invariants, typically written in OCL, for specifying the well-formedness rules. Following the UML profile spirit, a specialized metamodel strictly extends these two parts of a metamodel:

Definition 2. *A metamodel MM' is a specialization of a metamodel MM if MM' extends the structure and/or the invariants of MM. MM' is built by adding to MM, new meta-elements, new attributes in meta-elements, new operations in meta-elements and/or new associations between meta-elements without removing any existing elements of MM. MM' defines additional invariants without removing the existing ones of MM.*

The specialization of a family, that is the definition of a subfamily from a superfamily, is simply made by first specializing its metamodel. As this specialization is a strict extension and does not remove anything, all statements that are made about a superfamily also apply to all subfamilies. We lay down that subfamilies "inherit" execution operations, adaptation checks and adaptation actions from the superfamily. Anything that can be done (from an execution or adaptation viewpoint) with a model of the superfamily can also be done with a model of the subfamily.

Second, in addition to new elements (structural or invariants) in the metamodel, new attributes can be defined for a subfamily, that is, new execution operations, adaptation checks and adaptation actions. Concisely, a family specialization is defined as follows:

Definition 3. *A family F' is a specialization of a family F if the metamodel of F' is specializing the metamodel of F. F' inherits from all the attributes (execution operations, adaptation checks and adaptation actions) of F and can define additional ones.*

Multiple inheritance between metamodels and families is allowed. However conflicts are supposed to be avoided through a careful design.

As families can inherit from each other, it is then possible to define hierarchies of families. The root of a hierarchy is an i-DSML; dealing only with executable models without any adaptation concern. The root i-DSML family can be specialized to define either others i-DSML (without adaptation) or adaptable i-DSML (including adaptation). The more a family is placed at the bottom of the hierarchy, the more its metamodel is specialized and allows the definition of specific adaptation policies. Reaching a certain level of specialization, some adaptable i-DSML can even be based on specific business content as explained in the following subsection.

3.3 Domain versus Business Level Adaptation Policies

Another notion emerges from the previous ideas although it can be tricky to formalize it. Along with the generalization/specialization, a family may represent a set of business-neutral models or not. A business-neutral or domain-level scope expresses that an adaptation policy can be applied on any model conforming to the metamodel of the family, independently of its content (it is said to be "domain" because it is based only on the constructs of the *Domain-Specific Modeling Language*). Conversely, a business-level adaptation policy is based on specific business elements contained in the model. Domain-level families are generally placed on top of the hierarchy while business-level ones are placed in the bottom.

For example, constraining a process to have at least one fork is business-neutral and then situated at the domain level. Indeed, any process may potentially satisfy this fork constraint, regardless of its business content (a cooking recipe, a software development method, etc.), so that the associated adaptations can be reused across a large variety of models. Conversely, constraining a process to have an activity named "beat eggs" breaks the neutrality in the sense that it now presupposes that the process falls within the cooking domain. In that case, the adaptation written for such a family can be very accurate, but far from being reusable.

Technically speaking, the fringe between business-level and domain-level is somewhat fuzzy. However, we can say that if an adaptation semantics, within an adaptation check or an adaptation action, is based on literal values (such as the string value "beat eggs"), then the adaptation will be considered as business-level.

4 Putting *PDL* into the Framework

To give a better understanding of the ideas developed in the previous section, we reconsider the illustration of the *PDL* i-DSML from a family-based framework point of view. Fig. 4 defines a possible family hierarchy for our i-DSML. Each family is graphically described by a box with four compartments. They contain, from top to bottom: the name of the family that references the eponymous metamodel, the execution operations, the adaptation checks and the adaptation actions associated with the family. The hierarchy defines six families: one dedicated to execution only (*PDL*), one business-level family (*ManagedSkipAdaptPDL*) and four domain-level families. In order to distinguish these three kinds of family, an `<<execution>>`, `<<business>>` or `<<domain>>` conceptual stereotype has been placed above each family name. In this paper, we show only the metamodel of the family *DependSkipAdaptPDL* because, thanks to the inheritance, this metamodel contains all the elements defined in its superfamilies. It can then be used to describe the evolution of the metamodels along the hierarchy (excepting for the business-level family). This metamodel is represented on Fig. 5.

All specializations for the *PDL* i-DSML presented in the rest of this section are extending the structure of the metamodel. However, in many cases, restricting the possible models solely by addition of OCL invariants is sufficient to define adaptation policies. As an example, in [6], we study the adaptation of basic UML state machines in case of unexpected events. With a general state machine, no adaptation decision can be taken. However, imposing that a transition associ-

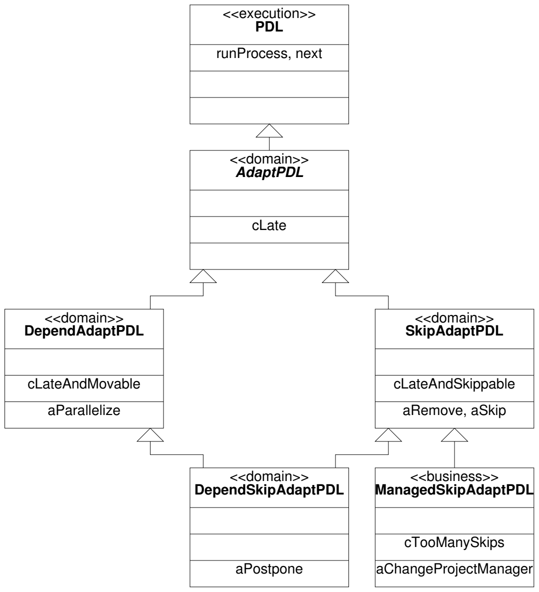

Fig. 4. Hierarchy of families for *PDL* adaptation

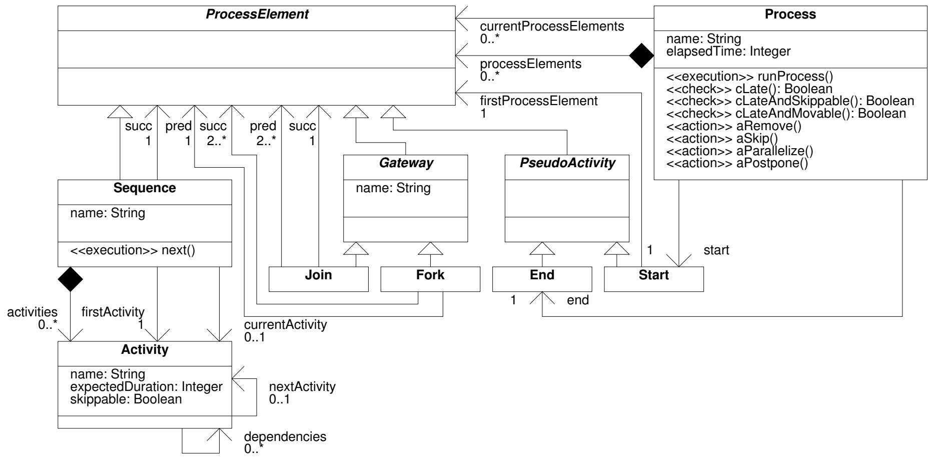

Fig. 5. The i-DSML corresponding to the *DependSkipAdaptPDL* family

ated with each expected event is starting from each state of the state machine leads to be able to determine if an event is expected or not (there exists or not an associated transition). Moreover, imposing that a given event always targets the same state leads to be able to automatically know how to add a state and transitions associated with the unknown event. These two restrictions are only defined through two OCL invariants without any modification of the metamodel, while they have a huge impact on the adaptation policies. Here lies the power of the concept of family.

4.1 Description of the *PDL* family

The root of the family is called *PDL*. It is the i-DSML presented in Sect. 2.1. As this i-DSML is only dedicated to execution, no adaptation checks nor adaptation actions are defined but only the execution operations (`runProcess()` of `Process` and `next()` of `Sequence`).

4.2 Description of the *AdaptPDL* family

Following the discussion of Sect. 2.3, a first basic extension of the *PDL* metamodel consists in adding an `elapsedTime` integer attribute to the `Process` meta-element and indicating the real execution time of a process. This leads to define the *AdaptPDL* family. Thanks to this attribute, it is now possible to determine if the process execution is late by comparing the expected duration with the real elapsed time since the beginning of the process. This checking is realized by the `cLate()` operation added to the `Process` meta-element and prefixed by a `<<check>>` conceptual stereotype as shown on Fig. 5. It also appears in the third compartment of the *AdaptPDL* family box on Fig. 4. However, as explained in the discussion of Sect. 2.3, there is no way to define adaptation actions with this metamodel yet. This will be done with the subsequent specializations adding new elements on the metamodel.

One may wonder why it is relevant to define a family for an adaptable i-DSML that does not define any concrete adaptation actions. The reason is that the attribute defined and the associated adaptation check, are shared by several subfamilies. These elements are then directly inherited in all these subfamilies without requiring to define them several times. Making an analogy with object-oriented programming, the *AdaptPDL* family can be seen as an "abstract family". For this reason, this family name has been italicized in Fig. 4.

4.3 Description of the *SkipAdaptPDL* family

As proposed in Sect. 2.3, in case of delay, an adaptation action can be to remove unnecessary activities in the rest of the process. In order to be able to catch up activities that can be removed, we need to mark them. So, we add a `skippable` boolean attribute to the `Activity` meta-element. This leads to define the *SkipAdaptPDL* family. The designer has now to express which are the skippable activities in its process definition.

Thanks to this attribute, in addition to know that we are late with the `cLate()` adaptation check, it is now also possible to determine if a next activity is skippable. This checking is thus twofold and is realized by the `cLateAnd-Skippable()` adaptation check added to the `Process` meta-element as shown on Fig. 5. Two adaptation actions have been defined. The first one removes skippable activities (i.e., it modifies the static part of the i-DSML). This adaptation action is drastic and maybe the designer would appreciate a softer solution. Then, instead of statically removing skippable activities, we propose another adaptation action that only skips them (i.e., it modifies the dynamic part of the i-DSML, updating the current activity to the benefit of a next activity). These adaptation actions are respectively realized by the `aRemove()` and `aSkip()` operations added to the `Process` meta-element. They are prefixed by an `<<action>>` conceptual stereotype. As these operations are adaptation actions, they appear in the fourth compartment of the *SkipAdaptPDL* family box on Fig. 4.

As an example, in the Fig. 3, bottom hand side, the current activity is `Design`. If for the implementation sequence that follows this activity, the `CoreTest` and `UITest` are marked as skippable, once the `CoreImpl` activity finished and if we are late, the `CoreTest` activity will be skipped and the next executed activity will be `UIImpl`.

4.4 Description of the *DependAdaptPDL* family

In Sect. 2.3, instead of removing unnecessary activities we suggest to parallelize some activities in order to decrease the time of the process. Obviously, we cannot select randomly activities that will be parallelized because an activity may depend on another. Consequently, this adaptation could be achieved only if we are aware of the dependencies between activities. In order to state these dependencies, we add a `dependencies` reference to the `Activity` meta-element. This leads to define the *DependAdaptPDL* family. From this reference, in addition to know that we are late with the `cLate()` adaptation check, it is now also possible to determine if a following activity is movable (taking into account its dependencies). This double checking is realized by the `cLateAndMovable()` adaptation check added to the `Process` meta-element as shown on Fig. 5. The corresponding adaptation action is to transform unique sequences into multiple sequences in parallel. This adaptation action is realized by the `aParallelize()` operation added to the `Process` meta-element as shown on Fig. 5.

Fig. 6 gives an example of this kind of adaptation. At the top, there is the model before adaptation. The dashed arrows represent the dependencies between activities. These dependencies have been defined by the designer. The number written between the parentheses after the process name indicates the elapsed time. We are currently late (350 instead of the expected 300) and some activities are movable (taking into account their dependencies). The bottom of Fig. 6 shows the corresponding model after adaptation. For example, for the implementation sequence, as `CoreTest` depends on `CoreImpl`, they must be part of the same sequence. This is the same for `UITest` and `UIImpl`. However, “Core” activities and “UI” activities have no dependencies between each other. This is why two subsequences have been created.

4.5 Description of the *DependSkipAdaptPDL* family

Another interesting adaptation could be to postpone an activity (i.e., to move an activity toward the end of the process) in order to execute it only if we are not

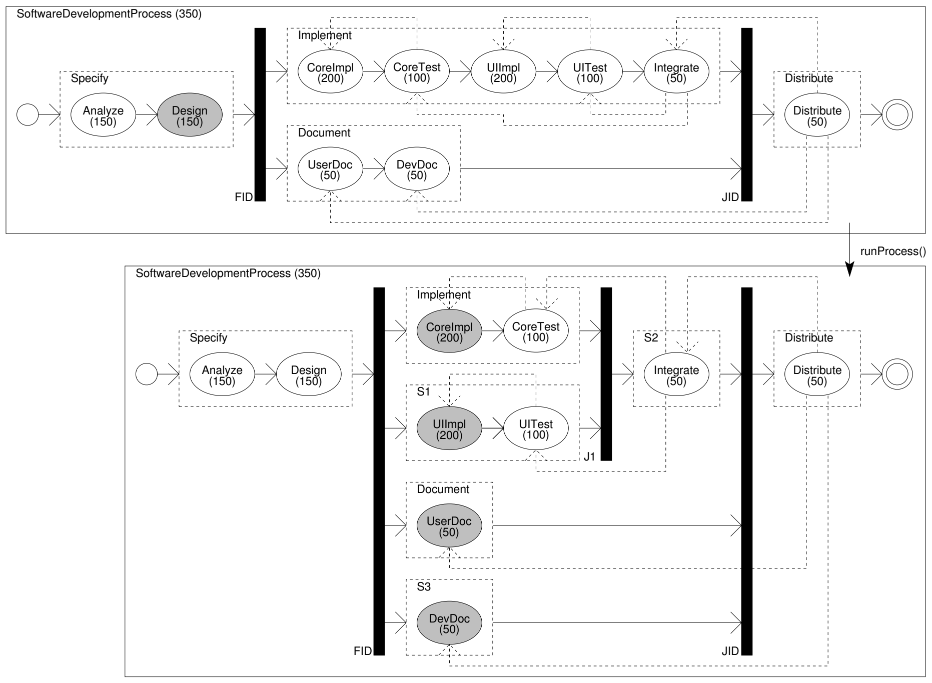

Fig. 6. The model before and after adaptation, conforming to the *DependAdaptPDL* family

anymore late. It is acceptable to postpone an activity if it is skippable (because it will possibly not be done) but it cannot be postponed later than an activity which depends on it. Consequently, we need at the same time the notion of skippable activity and the idea of dependencies between these activities. In order to have these pieces of information at runtime, we build the *DependSkipAdaptPDL* family that inherits from two superfamilies: *SkipAdaptPDL* and *DependAdaptPDL*. It means that the elements available in the i-DSML are the ones corresponding to the i-DSML of these two families. In this way, a complex checking can be realized by combining the `cLateAndSkippable()` and `cLateAndMovable()` adaptation checks. The corresponding adaptation action is to shift an activity as far as possible toward the end of the process. This adaptation action is realized by the `aPostpone()` operation added to the `Process` meta-element as shown on Fig. 5. Thanks to multiple inheritance, we are able to write an adaptation rule that is based on the merging of two families.

So far, all the previous families were exclusively at a domain level. As explained in Sect. 3.3, it means that for processing the adaptation, there is no need to have information about what is representing the instance of the `Process` meta-element in terms of business. It could be for example a software development process or a cooking recipe, as well. To give a concrete example of business-level adaptation, the next subsection presents a business-level family.

4.6 Description of the *ManagedSkipAdaptPDL* family

During the software development process, if we are really very late, a cause might be the project manager who is not skilled enough. Being very late may be defined by counting the skipped activities and specifying a maximum number of allowable skips. In this context, adaptation could be to fire the project manager and to hire a new one. This leads to define the *ManagedSkipAdaptPDL* family that is extending the *SkipAdaptPDL* family. This family defines a `cTooManySkips()` adaptation check that expresses if more than a given number (for instance three) of skips have already been done. The adaptation action `aChangeProjectManager()` of this family consists in creating a particular activity in the process: a *Change Project Manager* activity (this activity can be added in parallel of the existing activities of the process) which is immediately activated.

This family is at the business-level because we are aware that the instance of the `Process` meta-element will represent a project development with a manager. Such an activity will not make sense for all the processes defined with *PDL* i-DSML, such as a cooking recipe for example.

5 Related Work

In this article we highlighted some ideas coming from both the software architecture field and the MDE community in order to apply them to the recent concept of i-DSML.

Indeed, our inspiration is rooted in the works on architectural styles, a seminal research theme during the late 90s [10, 16, 17]. An architectural style defines a family of similar software architectures (e.g., client/server, pipe&filter, blackboard, etc.) and basically provides specific elements of design and rules to govern their arrangement. In [10] David Garlan highlighted a specialization relationship that may hold between styles (e.g., pipeline is a substyle of pipe&filter) so that some Architecture Description Languages (ADL) have supported this experimental feature, like ACME or ArchWare. Meanwhile, some authors have investigated how the concept of style could ease (self)-reconfiguration of systems at runtime [7, 14], based upon adaptation rules defined at the architecture-level and an adaptation loop very close to case (a) in Fig. 1, retrospectively speaking.

Likewise, Jim Steel *et al.* [18] proposed the model typing where, while conforming to a metamodel of course, a model may adhere to one or more types in the same way that an architecture described with an ADL may in addition satisfy to one or more styles. Logically, Clement Guy *et al.* [11] continued the works through the study of the subtyping relationship and more generally the influence of such a type-system at the model-level [19], but regardless to the adaptation issue.

6 Conclusion and Perspectives

In this paper, we have proposed a framework that enables the implementation of the direct adaptation of an executed model conforming to an adaptable i-DSML. An adaptable i-DSML defines models that are directly executable and adaptable. The framework relies on the concept of family and aims at properly arranging a number of elements of several natures that all serve the definition of adaptable i-DSML. A family gathers a given metamodel of an i-DSML with operations dedicated to its execution and adaptation. We have showed, through an example, that specializing a metamodel of an i-DSML enables to define more relevant and accurate adaptation policies. For this reason, families can inherit from each other allowing us to defining hierarchies of families, from the most general to the most specific. The inheritance offers conceptually the same advantages as in object-oriented programming such as the reuse of existing adaptation policies, the factorization of the same policies through a common superfamily or the specialization of existing adaptation policies. A family can be defined at a domain or business level depending on the fact that they are based on a particular business content or not. We applied this approach on a concrete example of a process model where several families have been built thanks to the family specialization.

The engine executing and adapting a model must currently contain hard-coded adaptation rules. Indeed, the execution operations, adaptation checks and adaptation actions are orchestrated and weaved through the code of the developer within the execution engine. However, it can be useful to modify this orchestration during the execution of the model. If a family offers several adaptation actions, one can be more suitable than another, according to the current context. To achieve this in a suitable way, as a short-term perspective of this work, we plan to define an i-DSML dedicated to the orchestration of the available operations for a family. An orchestration model will be interpreted by the execution engine in addition to the executed model. Concretely, this model will define an adaptation semantics (combinations of adaptation checks and adaptation actions) and its weaving with the execution operations. In addition, we can reach meta-circularity if we turn the orchestration i-DSML into an adaptable i-DSML as explained in this article. This unified approach can succeed because this orchestration model ought to be modified during the execution. In other words, as raised in [6], the adaptation semantics can be adapted at runtime and thus leading to a true meta-adaptation.

References

1. Business Process Modeling Notation (BPMN) Version 1.2. Technical report, Jan. 2009.
2. G. S. Blair, N. Bencomo, and R. B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.
3. E. Breton and J. Bézivin. Towards an understanding of model executability. In *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS '01)*. ACM, 2001.

4. E. Cariou, C. Ballagny, A. Feugas, and F. Barbier. Contracts for Model Execution Verification. In *Seventh European Conference on Modelling Foundations and Applications (ECMFA '11)*, volume 6698 of *LNCS*. Springer, 2011.
5. E. Cariou, O. Le Goaer, and F. Barbier. Model Execution Adaptation? In *7th International Workshop on Models@run.time (MRT 2012) at MoDELS 2012*. ACM Digital Library, 2012.
6. E. Cariou, O. Le Goaer, F. Barbier, and S. Pierre. Characterization of Adaptable Interpreted-DSML. In *European Conference on Modelling Foundations and Applications (ECMFA 2013)*, volume 7949 of *LNCS*, pages 37–53. Springer, 2013.
7. S.-W. Cheng, D. Garlan, B. R. Schmerl, J. a. P. Sousa, B. Spitnagel, and P. Steenkiste. Using Architectural Style As a Basis for System Self-repair. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, WICSA 3, pages 45–59. Kluwer, B.V., 2002.
8. P. J. Clarke, Y. Wu, A. A. Allen, F. Hernandez, M. Allison, and R. France. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 9: Towards Dynamic Semantics for Synthesizing Interpreted DSMLs. IGI Global, 2013.
9. B. Combemale, X. Crégut, and M. Pantel. A Design Pattern to Build Executable DSMLs and associated V&V tools. In *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. IEEE, 2012.
10. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. *SIGSOFT Softw. Eng. Notes*, 19(5):175–188, 1994.
11. C. Guy, B. Combemale, S. Derrien, J. Steel, and J.-M. Jézéquel. On Model Subtyping. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. S. Kolovos, editors, *ECMFA*, volume 7349 of *LNCS*, pages 400–415. Springer, 2012.
12. G. Lehmann, M. Blumendorf, F. Trollmann, and S. Albayrak. Meta-Modeling Runtime Models. In *Models@run.time Workshop at MoDELS 2010*, volume 6627 of *LNCS*. Springer, 2010.
13. OMG. Software Process Engineering Metamodell SPEM 2.0 OMG Draft Adopted Specification. Technical report, OMG, 2006.
14. P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*, pages 899–910. ACM, 2008.
15. M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4:14:1–14:42, 2009.
16. M. Shaw. Comparing architectural design styles. *Software, IEEE*, 12(6):27–41, 1995.
17. M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In *The Twenty-First Annual International Computer Software and Applications Conference (COMPSAC '97)*, pages 6–13. IEEE Computer Society, 1997.
18. J. Steel and J.-M. Jézéquel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007.
19. W. Sun, B. Combemale, S. Derrien, and R. B. France. Using Model Types to Support Contract-Aware Model Substitutability. In P. V. Gorp, T. Ritter, and L. M. Rose, editors, *ECMFA*, volume 7949 of *LNCS*, pages 118–133. Springer, 2013.