Variability management in the development of domain-specific languages for state machines

David Méndez-Acuña, Benoit Combemale, Benoit Baudry
DiverSE Team. INRIA and University of Rennes 1. France
{*david.mendez-acuna, benoit.combemale, benoit.baudry*}*@inria.fr*

Jérôme Le Noir
[2]Thales Research & Technology. France
jerome.lenoir@thalesgroup.com

Abstract — Domain-specific languages (DSLs) are becoming a successful technique in the construction of software intensive systems. Increasing the level of abstraction at which software solutions are expressed facilitates the participation of domain experts in the design and implementation of the system. In this context, the phenomenon of syntactic and semantic variability has started to appear within DSLs' specifications. When the same DSL is used in different application contexts, their diverse final users might require specialized language constructs and/or dedicated semantics. As a result, different variations of the same DSL need to be constructed and maintained.

Some of the system engineers in Thales Research & Technology have started to experiment this phenomenon when using state machine languages for specifying the behavior of the systems they build. The construction of each system supposes different modeling needs that, in many cases, are reflected in different, and possibly conflictive, semantic interpretations for the constructs of state machines language. As a result, there is an emergent need of engineering different variants of an in-house state machine language that fit the modeling needs for each situation. Of course, this fact demands additional (and quite important) engineering efforts that over-complexifies the DSL's development process.

In this document, we present our preliminary results towards a solution to this problem. Our solution is twofold. Firstly, we provide a language workbench that facilitates the development of DSLs that present syntactic and semantic variation points. This workbench uses some ideas from components-based DSL development and feature modeling. Secondly, we use our workbench to build a highly configurable DSL for modeling state machines. We report on the variation points supported by the DSL and we provide a tool demo that show the capabilities of the entire solution.

Keywords — Software languages engineering, domain-specific languages, language product lines.

1. Introduction

1.1. Purpose

VaryMDE is a bilateral collaboration (2011 - 2015) between the DiverSE team at INRIA and the MDE lab at Thales Research & Technology (TRT). This partnership explores variability management issues at both the modeling level and the metamodeling level (i.e., design and implementation of software languages). This document is the deliverable number eight (i.e., milestone 8) of the second issue and it presents a new version of the proposed technology including the feedback provided by Thales during last year.

1.2. Research context

The engineering of complex software intensive systems involves many different stakeholders, each with their own domain of expertise. It is particularly true in the context of systems engineering in which rather than having everybody working with code/model defined in general-purpose (modeling/programming) languages, more and more organizations are turning to the use of Domain Specific Languages (DSLs). DSLs allow domain experts to express solutions directly in terms of relevant domain concepts (*a.k.a.,* domain constructs), and use generative mechanisms to transform DSL specifications into software artifacts (e.g., code, configuration files or documentation), thus abstracting away from the complexity of the rest of the system and the intricacies of its implementation.

The adoption of DSLs has major consequences on the industrial development processes. This approach, a.k.a. Language-Oriented Programming [28], breakdowns the development process into two complementary stages (see Figure 1): the development, adaptation or evolution by *language designers* of one or several DSLs, each capitalizing the knowledge of a given domain, and the use of such DSLs by *language users* to develop the different system concerns. Each stage has specific objectives and requires special skills.

Figure 1 depicts the two interdependent processes that continuously drive each other's. The main objective of the language engineering process is to produce a DSL which tackles a specific concern encountered by engineers in the development of a complex system, together with its tooling. Once an appropriate DSL is made available to systems engineers, it is used to express the solution to this specific concern in the final system. It is worthwhile to note that, although this is unlikely in large companies, these roles can be alternatively played by the same people in smaller organizations.

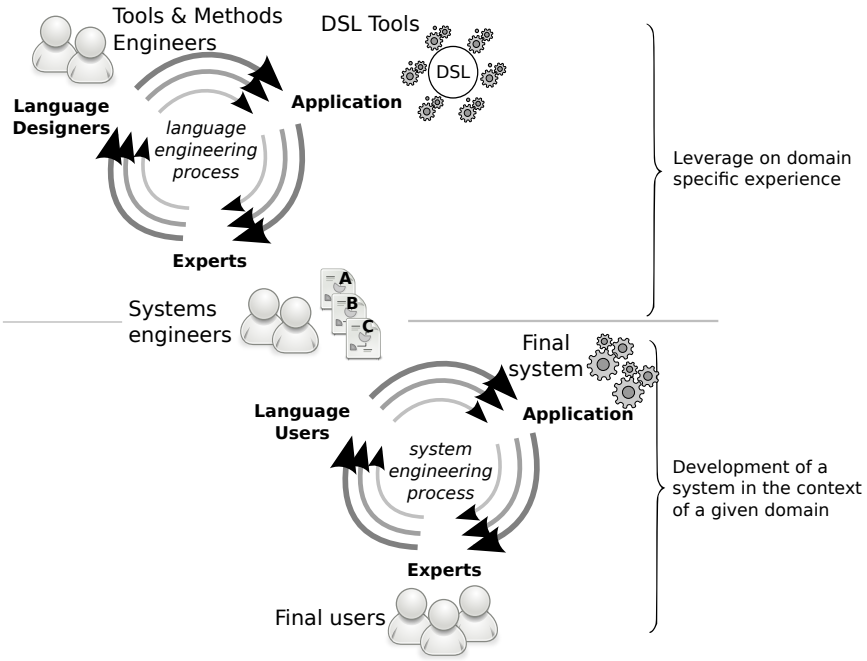

Fig. 1
Language engineering stakeholders

1.3. Problem statement

As a matter of fact, while DSLs have been found useful for structuring development processes and providing abstractions to stakeholders [15], their ultimate value has been severely limited by the cost of the associated tooling (i.e., editors, parsers, etc...). The construction of that tooling is a challenging and time-consuming task that requires specialized knowledge. A language designers must own not only quite solid modeling skills but also the technical expertise for conducting the definition of specific artifacts such as grammars, metamodels, compilers, and interpreters.

The situation becomes even more difficult in the context of multi-domain companies (such as Thales) where there are several domains that co-exist and overlap through different business units. In that case, diverse, and possibly conflicting, requirements often appear on the same DSL specially when it is required by two or more business units. According to its modeling needs, each business unit imposes specific constraints and demands particular capabilities. Language designers must face these situation by adapting the DSL and releasing different variants when needed.

The typical development process to address the aforementioned situation is quite similar to the one followed in the general case of software development (explained in [20]): at the beginning, there is a first DSL from which new development branches are forked and where the DSL is adapted to the new requirements. After some repetitions, language designers have a set of DSLs (*a.k.a* family of DSLs) that share some commonalities (materialized as replication of code) and that offer different capabilities that have been adapted for the application context. As one may expect, this approach introduces several problems. For example, because of code replication, all the effort invested in maintenance and bug fixing should be replicated as well.

In this context, it is desirable to have an approach that facilitates the engineering of families of DSLs. Such approach should be intended to exploit reuse as an alternative to code replication and to manage the variability among the different DSLs that compose the family. In addition, it is worth noting that an approach for dealing with families of DSLs has to take into account that the differences among members of a family of DSLs can be found in at any of the implementation concerns[1]. The work presented in [2] provides a classification of the possible types of variability that can be found within families of DSLs. A brief summary of this classification is introduced below:

- **Syntactical variability**: There is syntactical variability between two DSLs if they do not offer exactly the same language constructs. Among other situations, syntactical variability appears when the domains of the involved DSLs overlap. In that case, the DSLs share the constructs that represent the domains' intersection and differ in the constructs proper of each domain. For instance, a DSL for expressing state machines overlaps the domain of a DSL for expressing flow charts since they share the some constructs on

[1] A *DSL specification* is composed of three implementation concerns: abstract syntax, concrete syntax, and semantics [14]. The *abstract syntax* refers to the structure of the DSL expressed as the set of constructs that are relevant to the domain and the relationships among them. The *concrete syntax* refers to the association of the language constructs to a set of symbols (either graphical or textual) that facilitate the usage of the DSL. Finally, the *semantics* of a DSL refers to the meaning of its domain constructs.

boolean expressions. In the case of the state machines, the DSL includes these boolean expressions for expressing guards in the transitions whereas in the case of the flow charts they are used for expressing the conditions in the decision nodes.

- **Semantic variability:** There is semantical variability between two DSLs when, for the constructs they share, the implementation of the semantics is different. Among other situations, semantic variability appears when the semantics of the DSL depends not only on the domain knowledge but also on the application context. For instance, the DSL for state machines used for specifying the trains traffic might have some differences with respect to the DSL for state machines used for specifying the behavior of a watch.
- **Representation variability**: There is representation variability between two DSLs if, for the constructs they share, the implementation of the concrete syntax is different. Among other situations, representation variability appears when certain notation is more appropriated for a particular application context than another one. Consider, for example, the dichotomy between textual or graphical notations. Empirical studies such as the one presented in [22] show that, for a specific case, graphical notations are more appropriate than textual notations whereas other evaluation approaches argue that textual notations have advantages in cases where models become large [8].

The aforementioned dimensions of variability are not mutually exclusive. In fact, there are cases in which several types of variability appear at the same time in the same language product line. In such cases, an approach for multi-dimensional variability modeling [24] is required.

1.4. Variability in DSLs for state machines

Nowadays, there are several DSLs for expressing state machines. Each DSL provides a proper set of constructs (that are not always the same) and dedicated semantics. In other words, we can find syntactic, semantic, and representation variability in this set of DSLs. Accordingly, DSLs for state machines are a good example for families of DSLs. The study presented by Crane et al [4] illustrates this issue by comparing thee DSLs for state machines at the level of syntax and semantics. The main contribution of that research is a set of syntactic and semantic variation points existing between the three DSLs.

In the following, we illustrate the phenomenon of semantic variability in state machines by means of two toy examples. Our intention is to show how different modeling needs converge to different requirements in terms of the semantics.

1.4..1 Alarms and conflicting transitions

Fig 2 shows an state machine that specifies the internal behavior of a simple alarm. It is composed of two states *AlarmOn* and *AlarmOff* each one indicating whether the alarm

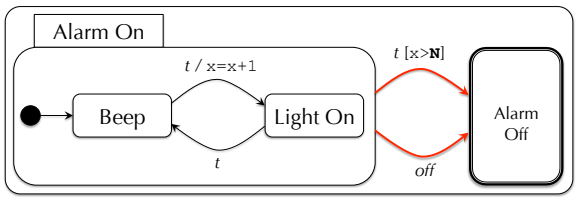

Fig. 2
State machine for an alarm. In case of conflicting transitions, the priority is given by **highest** scope.

is active or not. In the active state, the alarm oscillates between two states: *Beep* (which produces a 'beep' sound) and *LightOn* (which momentously turn on a light indicator). The rhythm of the alarm is defined by the event t that represents an automatic clock tick. The alarm is turned off in two cases: i) on arrival of the event *off*, indicating that some external entity gives the turn off instruction; and ii) when the time during the alarm has been activated arrives to a predefined limit (**N** clock ticks).

Intuitively, we can express this behavior by using two states: *AlarmOn* and *AlarmOff*. Besides, we can use two transitions for expressing the situations in which the alarm should be turned off. The first one reacts on the arrival of the event *off* whereas the second one reacts on the arrival of a new clock tick when the guard $x > N$ is true. Note that, in the second case, there is a conflict between the transitions inside the state *AlarmOn* and the transition from *AlarmOn* to *AlarmOff*. This conflict is due to the fact that the same event (i.e., t) is used for achieving two different situations. In order to achieve the expected behavior, the priority should be given to the transition with the highest scope. That is, the transition outgoing from the *AlarmOn* that is the higher state within in the states' hierarchy.

1.4..2 Positioning systems and conflicting transitions

Consider now the state machine presented in the Fig. 3 that specifies the behavior of a positioning system embedded in a given device. The system is intended to perform three tasks: i) to calculate the current position of the device; ii) to send the positioning information to the device itself so other parts of the system can use it; and iii) to send the device position to an external server that monitor the position of a pool of registered devices. The tasks i and ii should be performed constantly with a rhythm given by a clock tick represented by the event t. The tasks iii has to be performed every **N** clock ticks.

The state machine that specifies such behavior oscillates between the states *CalculatePosition* and *SendFeedback*. Changes between the states are specified in two transitions fired the event t. The *SendFeedback* state is composed of the local and the remote feedback representing the connection to the device and the server respectively. The remote feedback is only sent when the guard $x > N$ is satisfied. Similarly to the case of the alarm, when this guard is true there is a conflict between the internal transitions of the *SendFeed-*

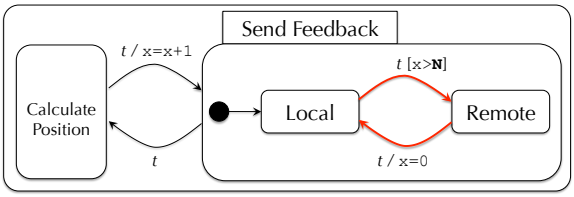

Fig. 3
State machine for a positioning system. In case of conflicting transitions, the priority is given by **lower** scope.

back state and the transition going from *SendFeedback* to *CalculatePosition*. Differently from the case of the alarm in this case the prirotiy should be given to the internal transitions. That means that in this case the priority should be given to the transition with lower scope.

1.5. Contributions

In this document we present our preliminary results towards the solution of the problem explained above. In particular, the contribution of this document is twofold. First, we provide an approach for engineering language product lines from existing families of DSLs. Second, we provide the implementation of a language product line for a family of DSLs for state machines. This family includes UML state diagrams [23], Rhapsody [12], and Harel's state charts [13]. In addition, we include Stateflow [21].

Scope: This approach is focused on supporting syntactic and semantic variability. Representation variability is out of the scope of this research.

1.6. Organization of the document

The remainder of this document is organized as follows. First, in section 2., we introduce our language workbench that supports the construction of families of DSLs. This description goes from the generalities of the approach to the specificities in terms of implementation details and the technological space it supports. Secondly, in section 3., we deeply describe the implementation of the family of DSLs for state machine using our workbench. To do so, we describe the syntactic and semantic variation points included in the family and we show how those variation points are implemented in our tool. Thirdly, in section 4., we present a tool demo that shows the entire solution (i.e., workbench + family of DSLs for state machines) in action.x Finally, in section 5. we conclude the document and present our perspectives.

2. The approach: From families of DSLs to Language Product Lines

In this section, we present our approach for dealing with the different types of variability existing within families of DSLs. Inspired from works such as [19], [27], and [29], among others, we propose to use the ideas of software product lines engineering (SPLE) to this end. To put it in another way, while software product lines are intended to deal with families of software products, we use *language product lines* for dealing with families of DSLs.

Our approach supposes the existence of a family of DSLs. That is to say, we suppose that there is a set of DSLs that share some commonalities and where it is necessary to manage the variability. The objective is to build a language product line that: (1) captures all the commonalities and particularities of the members of the family; (2) is implemented in a unique technological space; and (3) maximizes reuse during the development process. Naturally, the possible products of the language product line should correspond with the members of the family of DSL.

In order to achieve such objective, we follow the methodology illustrated in Figure 4 and deeply explained in the reminder of this section.

2.1. Step 1: Breaking-down the family of DSLs

The first step towards the construction of a language product line from a family of DSLs corresponds to separate the commonalities from the particularities existing within the involved DSLs. Both, commonalities and particularities, should be defined in separated language modules that can be later assembled. The definition of a given DSL corresponds to select and compose: (1) the modules that contain the commonalities the DSL shares with the other family members, and (2) the modules that contains its particularities.

For this reason, the design and the implementation of the common assets of a language product line require support for implementing DSLs in the form of interdependent language modules. Namely, it should be possible to specify a language module with a subset of a DSL specification that can be later composed with other language modules. As demonstrated in approaches such as MontiCore [18] or Neverlang [26], it is possible to build language modules in the same way that software modules are built. In that context, we have identified two different forms of modularization for the particular case of software languages: *separation of concerns* and *separation of features*. Let us explain each of this forms of modularization as well as introduce the tool support we provide in order to deal with them.

2.1..1 The notion of separation of concerns

Separation of concerns refers to the capability of decomposing the specification of a DSL into the different implementation concerns. In other words, the abstract syntax, the concrete syntax, and the semantics are implemented in different software artifacts. Note that this type of modularization is intended to support semantical and representation variability within families of DSLs. If we can specify each of these implementation concerns a different artifact, then we can have several implementations of the concrete syntax and semantics for the same abstract syntax. Besides, it

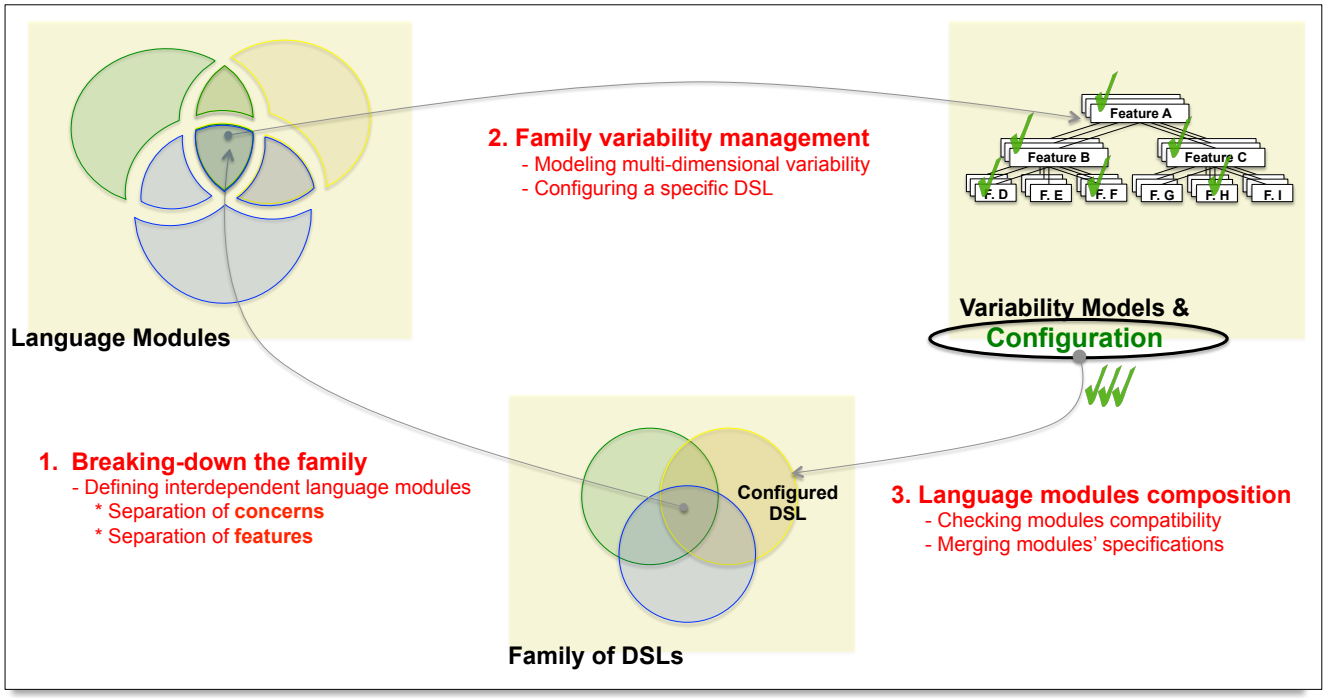

Fig. 4
Approach overview

would be possible to choose one or the other according to the needs in a particular application context. It is important to remember, however that we put into aside the problematics of dealing with concrete syntax. Our approach is focused on supporting the expression of abstract syntax and (operational) semantics.

Tool support. The tool support we provide to support separation of concerns relies on the possibility of defining the abstract syntax and the semantics of a DSL in different metalanguages. This idea is based on the discussion presented in [16] where the concept of *mashup of meta-languages* is introduced to refer to an artifact that specifies one implementation concern. In our case, we express the abstract syntax of the DSL in metamodels written in Ecore (the metamodeling languages offered by the Eclipse Modeling Framework (EMF)[2]) and where the operational semantics is specified in Xtend[3]. We use Kermeta 3 (K3)[4] as bridge for weaving the structure defined in the metamodel with the behavior specified in the Xtend.

Let us illustrate this idea by using a simple example. Consider the metamodel introduced at the top of Figure 5. It contains two classes `X`, and `Y`; The class `X` contains elements of type `Y` by means of the containment relation `yes`. In turn, the code snippet at the bottom of Figure 5 introduces some operational semantics to this metamodel by using K3. Note that the main feature of K3 is the notion of aspect that permits to weave the operational semantics defined in a Xtend class to a metamodel defined in Ecore.

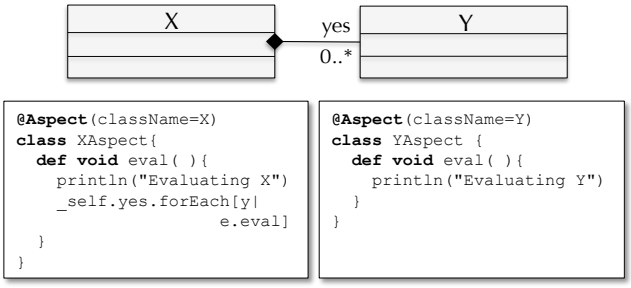

Fig. 5
A simple metamodel

In our example, the metaclass `X` is enriched with the operation `eval()` that contains a loop that sequentially invokes the operation defined for the class `Y`. This operation is also defined by using one aspect; in this case extending the metaclass `Y`. Note that the language designer is free to name the aspects as he/she considers correct. In this example, we use the convention `Aspect+<<metaclass.name>>`.
We use Melange[5] to integrate and execute the definitions of the abstract syntax and semantics of a DSL. Melange is a language for modeling in the large that facilitates the integration of the different artifacts that compose the specifica-

[2]EMF website: `https://eclipse.org/modeling/emf/`
[3]Xtend website: `https://eclipse.org/xtend/`
[4]K3 website: `https://github.com/diverse-project/k3`
[5]Melange website: `http://melange-lang.org/`

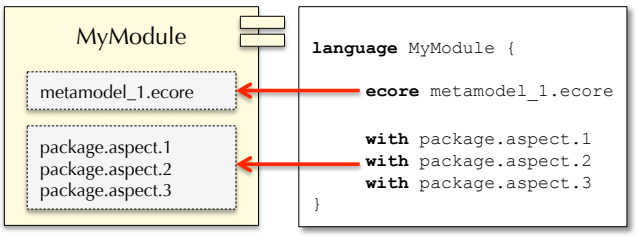

Fig. 6
Using Melange for weaving metamodels and aspects

tion of a DSL. Figure 6 illustrates the use of Melange. At the left we have an abstract representation of a language that is composed of a metamodel and three aspects implementing the semantics. At the right of the figure we have the corresponding Melange script.

2.1..2 The notion of separation of features

Separation of features refers to the capability of encapsulating subsets of language constructs in separated and interdependent language modules. Different modules specify different constructs; a complete DSL is obtained by composing a set of modules. As one might expect, the notion of separation for concerns is intended to support syntactic variability.

Note that both DSLs and modules are, at the end, set of language constructs. Hence, one may think that a module is also a DSL. Although this is technically true, there is a substantial difference between DSLs and modules. A DSL is a closed set of constructs that materializes a complete DSL specification that is ready to be used. Contrariwise, a module is set of constructs whose specification may depend on other constructs defined in other modules. A module can be used (and considered itself as a DSL) as long as their dependencies are fulfilled.

Accordingly, the main requirement for supporting separation of features in DSLs relies on the capability of expressing dependencies between language modules. In this context, we have identified two types of dependencies: *aggregation* and *extension*. In the following, we explain each of them and we present the corresponding tool support.

Language modules *aggregation*: In aggregation, there is a *requiring module* that uses some constructs provided by a *providing module*. The requiring module has a dependency relationship towards the providing one that, in the small, is materialized by the fact that some of the classes of the requiring module have references (simple references or containment references) to some constructs of the providing one.

In order to avoid direct references between modules, we introduce the notion of interfaces for dealing with modules' dependencies. In the case of aggregation, the requiring language has a *required interface* whereas the providing one has the *provided interface*. A required interface contains the set of constructs required by the requiring module which are supposed to be replaced by actual construct provided by other module(s).

It is important to highlight that we use *model types* [25] to express both required and provided interfaces. As illustrated on top of Figure 7, the relationship between a module and its required interface is *referencing*. A module can have some references to the constructs declared in its required interface. In turn, the relationship between a module and its provided interface is *implements* (deeply explained in [6]). A module implements the functionality exposed in its model type. If the required interface is a subtype of the provided interface, then the provided interface fulfills the requirements declared in a required interface. Note that the partial sub-typing relationship defined in [10], permits a required interface being partially fulfilled by a provided interface. The result of the composition will be a module with a new required interface that contains only those elements that were not provided.

Language modules *extension*: In this case, there is an extension module that (naturally) extends the functionality provided by *base module*. The extension module has a dependency to the base module. Moreover, the extension module has little sense by itself without the existence of a base module [9]. Note that this is a conceptual difference with respect to modules aggregation where the required make sense by itself but requires some external services in order to work correctly.

There are to different mechanisms for extending a base module: *constructs specialization* and *open classes* [3]. In constructs specialization, the constructs of the base module can be extended by adding new subclasses. The extension module contains the new subclasses that reference (by means of the inheritance relationship) the constructs of the base module that are being extended. In this case, the base module remains intact in after the composition but there are additional constructs. In open classes, constructs of the base module can be re-opened and modified by the extension module. For example, for adding a new attribute to a given construct without creating a sub-class, or for overriding a given segment of the semantics. In this case, the extension module is altered after the composition phase.

Similarly to aggregation, dependencies between the base and the extension modules are specified through interfaces. The base module exposes an *extension point interface* with the constructs that can be extended. In turn, the *extension interface* declares the constructs of the base module that are being extended.

Like in aggregation, and as illustrated at the bottom of Figure 7, we use model types of expressing these interfaces. Although the approach is quite similar, there is one fundamental difference between the interfaces in aggregation and the interfaces in extension: the relationship between an extension module and its extension interface is *usage* more than just referencing. That means that the module can to reference the elements declared in the required interface and also modify them by adding new elements. This capability is introduced to support extension by the open-classes mech-

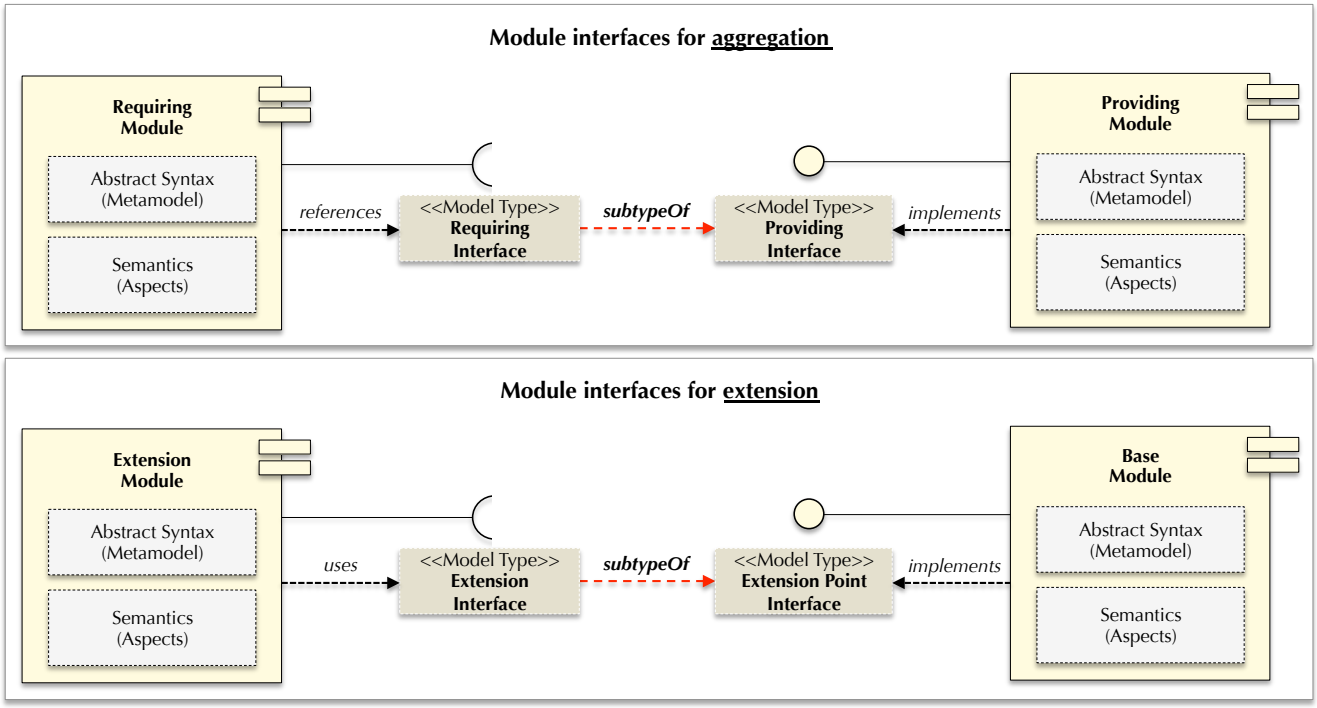

Fig. 7
Interfaces for modularization of DSLs

anism.

Tool support: The tool support we provide for specifying interfaces on language modules is based on annotations at the level of the abstract syntax (i.e., the metamodel). That means that the required constructs of a requiring module are declared as meta-classes in the metamodel as the same as the actual constructs of the module. However, the required constructs are annotated with `@Required` to distinguish them from the actual constructs. A similar approach is done for specifying the extension interface, this time, the corresponding annotation is `@Extends`.

Based on those annotations, we provide a tool set that extracts the corresponding model types that, later, will permit to perform compatibility checking and modules composition.

2.2. Family's variability management

Once the family of DSLs is completely modularized, the next step is to build a variability model that expresses the commonalities and particularities existing within the family members. The objective of this variability model is to offer a configuration mechanism that permits to derive a particular DSL based on these differences. This variability model should ensure that the possible configurations correspond to combinations of language modules that, once assembled, will produce valid DSLs. To this end, the variability model should rely in a formalism that allows the expression of these combination rules. Because feature models (FMs) [17] became the *"de facto"* standard to express common and variable features in a software product line [1], we will use them for expressing variability in language product lines.

2.2..1 Multi-dimensional variability modeling

As mentioned in the introduction, an approach for variability modeling in language product lines should take into account that variability in DSLs may appear in several implementation concerns. Since the scope of this approach is restricted to abstract syntax and semantics, we propose a variability modeling solution that includes these two dimensions while putting into aside representation variability.

Currently, we can find some ideas for dealing with multi-dimensional variability. As discussed in, [24] there are, at least, two strategies. Firstly, each dimension of the variability can be represented as a direct child abstract feature of a variability model that represents the variability of all the product line. Alternatively, each dimension of the variability can be represented in a separated variability modeling. Constraints between features of different dimensions are implemented as cross-models relationships.

For simplicity, we decided to follow the first approach. Dealing with several variability models and cross-cutting relationships might over-complexify the tooling since language designers will have to deal to more development artifacts and be sure that the corresponding referencing is correct.

Other important requirement of the variability modeling

approach is related to the mapping of the features in the variability model and the language modules that actually contain the specification of the DSL. Our approach is based on the fact that syntactical variability is supported by separation of features and that semantical variability is supported by separation of concerns. So, we associate syntactical features to language modules and semantical features to the aspects that implement the semantics of each modules.

Figure 8 illustrate our approach with a simple example. It is composed of two syntactical features i.e., feature **A** and feature **B** (at the left of the variability model) that are mapped to their corresponding metamodels. Note that the feature B requires of feature A. This is expressed as a constraint at the level of the variability model and materialized by means of modules interfaces in the modularization approach. On the other hand, there are two semantical features in the first level. These features correspond to semantical decisions on modules B and A respectively. Indeed, features **3** and **4** are mapped to different semantics implementations of the module B. The same occurs in the case of features **5** and **6** with the module **A**.

Tool support: The tool support we provide for multi-dimensional variability is based on the Common Variability Language (CVL). Language designers should design the variability model according to the principles explained below.

2.2..2 DSLs configuration

Once the variability of the language product line is correctly specified, the next step is to configure DSLs by using the variability model. Since the variability model is expressed in CVL, the configuration of the language product corresponds to the specification of a realization model that captures the decisions made by the language designers that are configuring the DSL. Our approach uses the realization model to produce the corresponding Melange script.

As an example, consider the configuration presented in the variability model of the Figure 8. The corresponding Melange script is presented in the following listing code snippet:

```
language ModuleA {
   ecore MetamodelA.ecore
   with package.A.Semantics6
}

language ModuleB {
   ecore MetamodelB.ecore
   with package.B.Semantics4
}

language MyDSL {
   aggregation(ModuleB, ModuleA)
}
```

Note that the Melange script only contains the language elements that correspond to a given configuration. For example, the ModuleA contains only the Semantics6 because it was the choice made at configuration time. Similarly, ModuleB only contains Semantics4. Note also that there is a third language appearing in the script: MyDSL. This language represents the composition of the language modules and can be understood as the root of the script. In this case, this statement of Melange indicates that the modules A and B are composed by aggregation. The first element in the operation corresponds to the requiring module and the second element corresponds to the providing module.

2.3. Language modules composition

Once the configuration process produces a Melange script that captures the choices made by the language designers for a particular DSL, it is necessary to compose the declared language modules and produce the DSL. The composition of a set of modules requires a previous phase of compatibility checking. Not all language modules are compatible and in that case composition cannot be performed. In our approach, check the compatibility of two language modules is to verify the sub-typing relationship between the required and provided interface (for the case of aggregation), and the extension and extension point interfaces (for the case of extension).

Once this compatibility checking is correctly verified, language modules are composed. In particular, their specifications are be merged to generate a complete language specification. This merging basically replaces the elements of the required interface by its corresponding implementation in the provided component. A similar process is performed in the case of extension.

3. Applying the approach: From a family of DSLs for state machines to a language product line

In this section we illustrate the applicability of our approach the case of a family for DSLs for state machines. Concretely speaking, we illustrate each of the elements of the approach presented in Figure 4 from the perspective of the state machines case study. We first describe the family, then we present the process we follow for breaking down the family. Finally, we show the variability model.

3.1. The family of DSLs for state machines

Figure 9 illustrates our family of DSLs for state machines. As aforementioned, this family is based on the work of Crane et al. [4]. Hence, the family includes UML state diagrams [23], Rhapsody [12], and Harel's state charts [13]. In addition, we include Stateflow [21].

3.1..1 Basic constructs

States, transitions, triggers, and guards: In their simplest form, state machines are graphs where nodes represent states

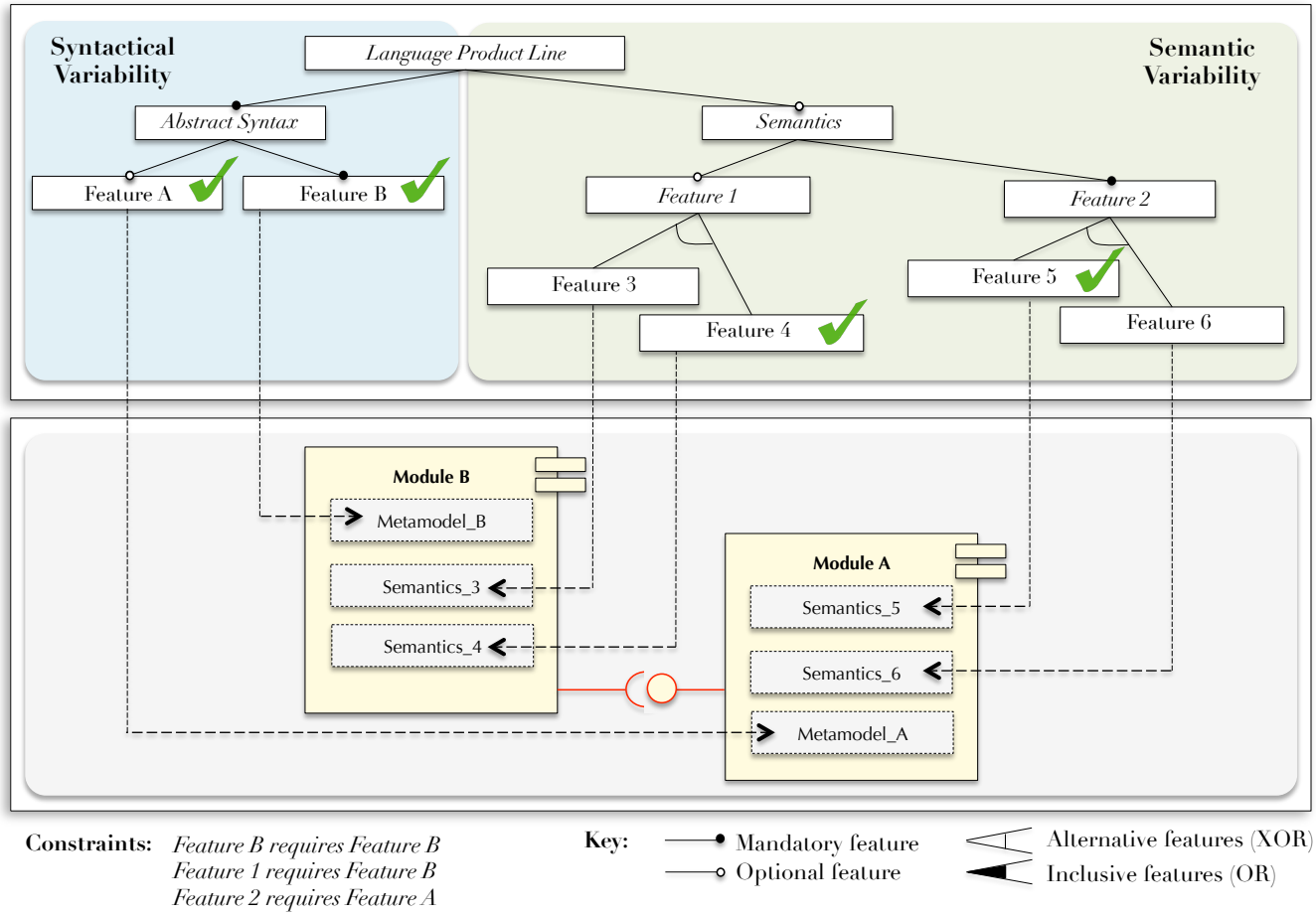

Fig. 8
A simple example for multi-dimensional variability and the corresponding mapping with language modules

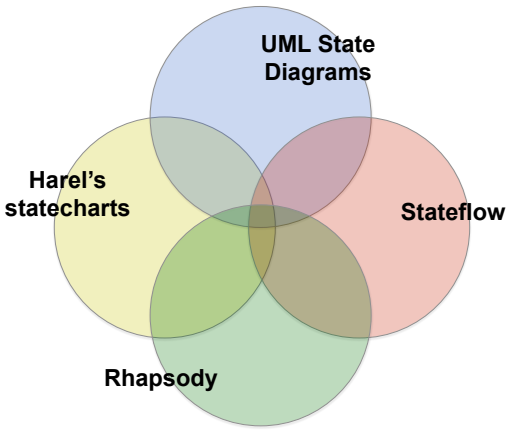

Fig. 9
A family of DSLs for state machines

and arcs represent transitions between the states [11]. The execution of a state machine is performed in a sequence of *steps* each of which receives a set of events that the state machine should react to. The reaction of a machine to set of events can be understood as a passage from an initial configuration (t_i) to a final configuration (t_f). A configuration is the set of active states in the machine.

The relationship between the state machine and the arriving events is materialized at the level of the transitions. Each transition is associated to one or more events (also called triggers). When an event arrives, the state machine fires the transitions outgoing from the states in the current configuration whose trigger matches with the event. As a result, the source state of each fired transition is deactivated whereas the corresponding target state is activated. Optionally, guards might be defined on the transitions. A transition is fired if and only if the evaluation of the guard returns true at the moment of the trigger arrival.

The initial configuration of the state machine is given by a set of initial pseudostates. Transitions outgoing from initial pseudosates are fired automatically when the state machine is initialized. In turn, the execution of a state machine continues until the current configuration is composed only by

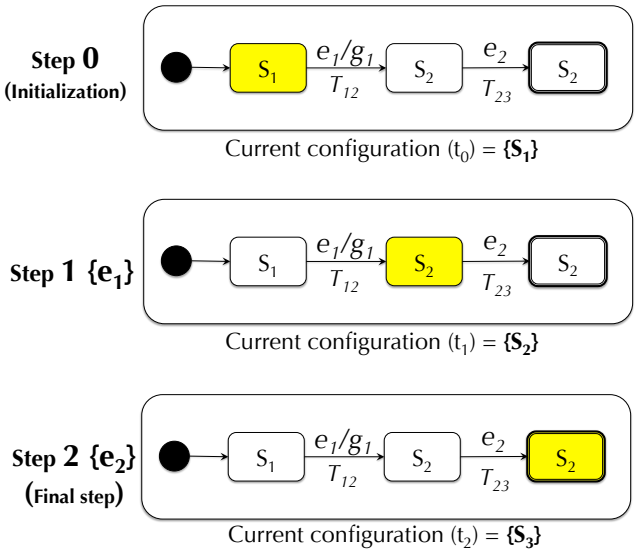

Fig. 10
Example of a simple state machine and its execution

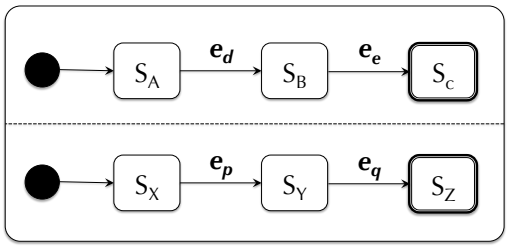

Fig. 11
Example of a simple state machine with two regions

final states (an special type of states without outgoing transitions).

Figure 10 illustrates the aforementioned behavior with a simple state machine. It is composed of the states S_1, S_2, and S_3. The state S_1 is the entry point of the state machine. This is expressed by means of the initial pseudosate (the filled circle) containing a transition towards S_1.When the event e_1 arrives, the state machine evaluates the guard g_1 and, if the evaluation returns true, the transition T_{12} is fired. Then, the state S_1 is deactivated and the state S_2 is activated. In the second step, event e_2 arrives and the final state S_3 is activated after firing the transition T_{23}.

Regions: All of the DSLs for expressing state machines that conform our family support the notion of region. A state machine might be divided in several regions that are executed concurrently. Figure 11 illustrates a state machine with two regions. Note that each region of the state machine contains its proper initial pseudostate.

Actions: Note that a state machine has some additional capabilities with respect to the ones introduced so far. States can define entry/do/exit actions. Similarly, transitions can have some effects. States' actions and transitions' effects

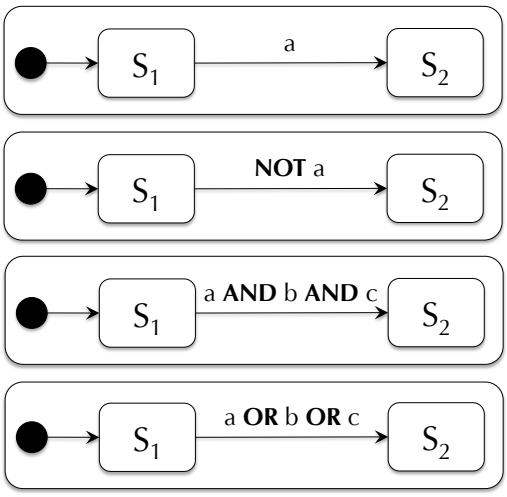

Fig. 12
Example of a triggers and triggers' combination

are intended to interact with the environment of the state machine. The environment is a mapping that associates variables with values.

3.1..2 Syntactic variation points

Let us introduce the syntactic variation points present in the family of DSLs for state machines. These semantic variation points are presented by showing the constructs available in the language and their availability in each of the involved DSLs

Triggers. Disjunction, conjunction, and negation: The first syntactic variation point supported by our tool refers to the relationship between triggers and transitions. While in section 3.1..1 (basic constructs) we introduced the idea that a transition is associated to certain event (first state machine in Fig. 12), it is also true that this relation can be more complex.

- **Triggers' negation:** A transition can be associated to the negation of an event. As a result, it will be fired in every step where its target state belongs to the current configuration and the associated event *does not arrive*. The second state machine in the Fig. 12 illustrates this situation.
- **Triggers' combination:** DSLs for state machine languages often offer the capability of associating multiple triggers to a transition. Multiple triggers can be combined by means of the OR and AND logic operators as shown in the third and forth state machines of Fig. 12.

DSLs configuration: In terms of triggers combination, Rhapsody is the more restrictive DSL where transition can be only associated to one event. Conversely, in UML a transition can be triggered by several events associated to the AND logical operator. In Stateflow the relationship among

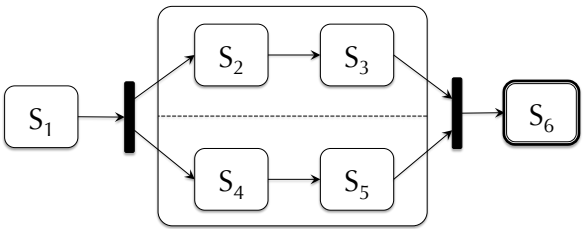

Fig. 13
Example of Fork/Join

multiple events in a transition is treated by means of the OR logical operator. Harel's state charts are even more expressive and allow conjunction and negation of events.

Pseudostates: Pseudo-states are special types of states that allow the expression of compound transitions. For example, as illustrated in Fig. 13 the pseudo-state `Fork` enables the bifurcation of a transition so different states can be executed in parallel. Similarly, the pseudostate `Join` enables the unification of two transitions outgoing from concurrent states. There are other types of pseudostates such as the history ones that stores a reference to the last sub-state executed in a composite state. Conditional pseudostates are also available enabling different execution paths for the state machine according to the value of the variables in the environment.
Formalisms configuration: There are certain differences in the DSLs regarding the pseudostates they support. All the tree DSLs offer the initial pseudo-states, as well as fork, join, junction and deep history. Shallow history is only supported by UML and Harel's state machines. A particular pseudo-state for representing dynamic choice is offered by UML. Static choice is supported by Rhapsody and Harel's state charts.

3.1..3 Semantic variation points

The reminder of this section presents the semantic variation points existing in our family of DSLs for state machines.

Events dispatching policy: The first semantic difference in the operational semantics of state machines refers to the way in which events are consumed by the state machine. In a first interpretation, simultaneous events are supported whereas in a second interpretation the state machine follows the principle of run to completion. Let us discuss each scenario in detail.

- **Simultaneous events:** There are DSLs for state machines that support this capability so a step consumes a subset of the events where the size of the subset can be greater than one and attends all of them at the same time. For example, consider the state machine presented in Fig. 14. In that case, the events e_1 e_3 arrive at the same time in the step 1 and they are attended simultaneously. As a result, the configuration of the machine after that step is S_2, S_5.

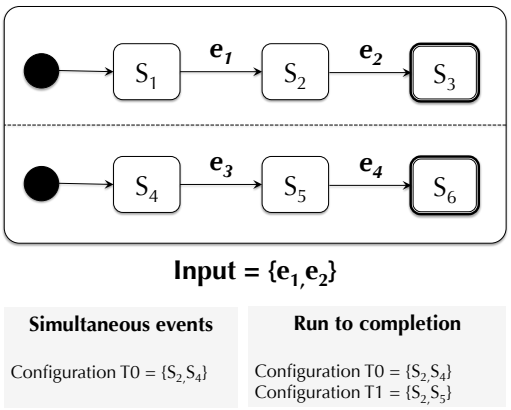

Fig. 14
Difference between simultaneous events and run to completion

- **Run-to-completion:** A different interpretation of this variation point is to comply to the run to completion policy. That means that the state machine is able only of supporting one event by time and it cannot consume another one until de execution of the current event is completed. For example, consider the state machine presented in Fig. 14. Differently from the later case, in this one the events e_1 e_3 that arrive at the same time in the step 1 are attended one by one. As a result, the configuration of the machine after that step is S_2, S_4 and an automatic additional step is dispatched then. After that second step, the configuration of the machine is S_2, S_5.

Note that in the example we presented for the sequential events interpretation, we show that the state machine executes the events in a given order that corresponds to the order in which the events are listed. However, it is important to clarify that it is not always like that. In fact, there is no notion of order in the consumption of events that arrive in the same step. This fact has been identified as a potential semantic variation point for state machines. However, it is out of the scope of this document.

DSLs configuration: The semantics of UML, Rhapsody, and Stateflow fit the run to completion policy for events dispatching whereas Harel's statecharts support simultaneous events.

Execution order of transitions' effects: It is possible to define actions on the transitions that will affect the execution environment where transitions are fired. These actions are usually known as transitions effects. All the DSLs for state machines in our family support the expression of such effects. However, there are certain differences regarding their execution.

- **Effects executed sequentially:** The first and most common way of executing the effects of a transition is by following the order in which they are defined. This is due to the fact that transitions effects are usually de-

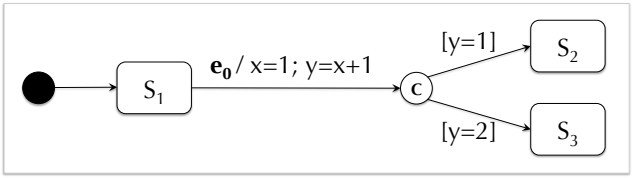

Fig. 15
Example of a state machine with effects in the transitions

fined by means of imperative action script languages where the order of the instructions is intrinsic. Under this interpretation, once the event e_0 is dispatched, the state machine in Fig. 15 moves from state S_1 to the state S_3. This is because the value of the variable y depends on the value of x that has been previously modified in a precedent instruction.

- **Effects executed in parallel:** The second interpretation to the execution of transitions' effect is to execute them in parallel. In other words, the effects are defined as a set of instructions that will be executed at the same time so no assumptions should be made with respect to the execution order. Under this interpretation, once the event e_0 is dispatched, the state machine in Fig. 15 moves from the state S_1 to the state S_2 Although the value of y depends on the value of x, the instructions are executed simultaneously.

DSLs configuration: UML, Rhapsody, and Stateflow execute the transition effects in parallel. Harel's statecharts execute transition effects simultaneously.

Priorities in the transitions: Because several transitions can be associated to the same event, there are cases in which more than one transitions are intended to be fired in the same step. In general, all the DSLs for state machines agree in the fact that all the activated transitions should be fired. However, this is not always possible because conflicts might appear. Consider for example the state machine presented in Fig 16. The transitions T_D and T_E are conflictive because they are activated by the same event i.e., e_2, they exit the same state, and they go to different target states. Then, the final configuration of the state machine will be different according to the selected transition.

In order to tackle such situations, it is necessary to establish policies that permit to solve such conflicts. Specifically, we need to define a mechanism for prioritizing conflicting transitions so the interpreter is able to easily select a transition from a group of conflicting transitions. One of the best known semantic differences among DSLs for state machines is related with these policies. In particular, there are two different mechanisms for solving this kind of conflicts:

- **Priority for deepest transitions:** A first mechanism for solving conflicting transition is to select the transition with the lower scope. That means, the transition

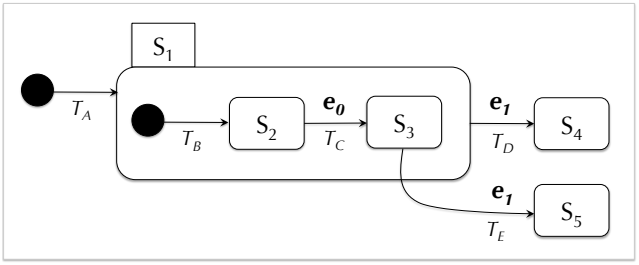

Fig. 16
Example of a state machine with conflicting priorities

that is deeper in the hierarchy of the state machine. In the example presented in Fig 16 the dispatched transition according to this policy would be the transition T_E so the state machine would move to the state S_5.

- **Priority for higher transitions:** The second mechanism for solving conflicts in the transition is to select the transition with the higher scope. That is, the transition in the higher level of the hierarchy of the state machine. In the example presented in Fig 16 the dispatched transition according to this policy is the transition T_D so the state machine will move to the state S_4.

DSLs configuration: The semantics of UML and Rhapsody fits on the first interpretation i.e., deepest transition priority whereas the semantics of Harel's statecharts and Stateflow fits on the second interpretation i.e., highest transitions priority.

Junction Pseudostate: The junction construct is a pseudostate that permits to create compound transition paths between two states [23]. Then, a transition can be defined in terms of several *segments*. All the DSLs for state machines studied in this document support this construct. However, there are various semantic interpretations so the use of the construct is different in each case. There are two fundamental differences explained below:

- **Events owner:** The first difference in the definition of a junction in a state machine refers to whether the associated events are specified in either the incoming or the outgoing transitions. These cases are illustrated in Figures Fig. 17**.1.a** and 17**.1.b** respectively.

  This semantic difference supposes different implementations for the static semantics. The validation phase of a state machine is intended to guarantee that a junction pseudostate respects the events definition. For example, for the first interpretation (events defined in the incoming transitions) the validation phase rejects junctions where the events have been defined in the outgoing transitions. In that sense, the state machine presented in Fig 17**.1.b** is considered ill-formed in DSLs whose semantics fit the second interpretation.

In addition, the operational semantics of an step in the state machine should be defined differently for each interpretation. In the case of events defined in the incoming transitions, the interpreter first searches for the current active transitions according to the dispatched events and the satisfied guards. After that, there may be junction pseudostates included the current configuration of the state machine. If so, the outgoing transitions of the junction pseudostates are automatically dispatched in the same step. A step in a state machine only will finish when there will not exists any junction pseudostate in the current configuration.

Fig. 17**.4** illustrates this semantics with a simple state machine. The `step_0` initializes the state machine and the first active state is `S1`. The `step_2` dispatches the event e_0 and activates the transition arriving to the state `S2`. The `step_2` dispatches the event e_1 and the junction is activated. In the same step the transition outgoing from the junction is fired so the final configuration after the `step_2` is the state `S3` (in bold in the figure). Finally, the `step_4` dispatches the event e_4 and activates the state `S4`.

In the second interpretation –where the events are defined in the outgoing transitions of the junction pseudostate– the activation of the transitions is different. A transition is activated not only when the associated event is dispatched but also when its target is a junction pseudostate. This is illustrated in Fig 17**.3** with a simple state machine. The `step_0` initializes the state machine and the first active state is `S1`. The `step_1` dispatches the event e_0 and activates the transition arriving to the state `S2`. At that point the state machine realizes that the transition outgoing from the state `S2` arrives to a junction. So, the interpreter fires that transition and the final configuration after that step corresponds to the junction. The `step_2` dispatches the event e_1 and activates the state `S3`. Finally, the `step_4` dispatches the event e_4 and activates the state `S4`.

DSLs configuration: The semantics of both UML and Stateflow fit the first interpretation whereas the semantics of Harel's statecharts and Rhapsody fit the second one.

- **Multiple outgoing transitions:** The second difference in the implementation of the junction pseudostates is related to the possibility of having several outgoing transitions. In one interpretation, multiple transitions outgoing from junction pseudostates are allowed so a transition can be bifurcated. In a second interpretation this is not possible and junction pseudostates can have one and only one outgoing transitions. Note that, as illustrated in Fig. 17**.2**, the way un which a transformation bifurcation is defined depends also on the interpretation that the DSL gives with respect to the events owner.

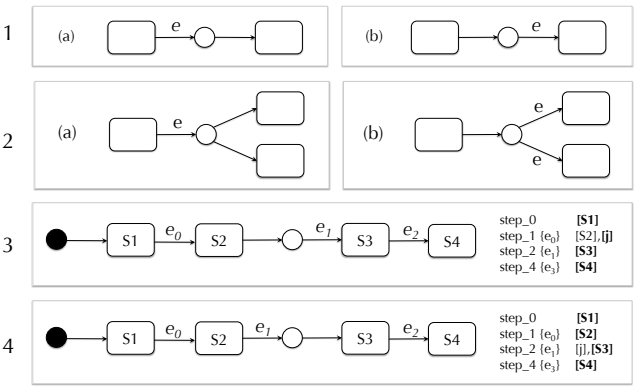

Fig. 17
Different examples for the semantics of the Junction pseudostate

DSLs configuration: The semantic of UML, Harel's statecharts, and Stateflow allow multiple outgoing transitions in junction pseudostates. Rhapsody does not support such functionality.

3.2. Breaking-down the family of DSLs for state machines

Figure 18 presents the results for the breaking-down phase in the case of the family of DSLs for state machines. Concretely, the figure shows the set of language modules in which we decompose the family. The main objective of this modular design is to support the variability points. However, we include some additional modules intended to facilitate the integration of new languages to the family. In the following, we explain in detail the proposed design:

- There is a first modularization decision regarding the identification and definition of a core module (i.e., **CoreFSM**). It contains the constructs that are included in all the DSLs of the family under study. In other words, this module encapsulates the commonalities among the family members. As a result, it is included in all the products that might be configured in the language product line.
- In the general case, the definition of a state machine requires an action language for expressing both, states' actions, and transitions' effects. As a result, DSLs for state machine usually define their proper action language. Not all the action languages provided in these DSLs have the same capabilities. In fact, these action languages are usually toy languages that are not expressive enough. In order to homogenize this part of the family, we separate the constructs of the action language from the constructs of the state machines. Hence, we define module (i.e., **ActionLanguage**) that is referenced from the core module. The dependency relationship between the core and the action language is an aggregation because the core *requires* the constructs defined in the action language.

- Note that the same analysis is valid for the case of the constraints language. The definition of a state machine requires a constraints language for expressing guard in the transitions. As a result, many constraints languages with different levels of expressiveness appear. We homogenize these languages in our language product line by defining a language module (i.e., **ConstraintsLanguage**) specialized in the definition of constraints. The dependency relationship between the core and the constraints language is an aggregation because the core *requires* the constructs defined in the constraints language.
- The next modularization decision we propose is intended to support the syntactic variation points regarding the different types of triggers supported by the different DSLs for state machines (see section 3.1..2). Since all the DSLs of the family support the notion of single trigger on transitions, this constructs is included in the core. However, more complex expressions such as negation, disjunction, or conjunction of triggers are not supported for all the DSLs. Then, we modularize each type of trigger in a separated language module (i.e., **AndTrigger**, **OrTrigger**, **NotTrigger**). Each type of trigger is implemented as a subclass of the constructs trigger. Hence, each of those triggers' modules can be seen as extensions to the core i.e., the specialize the construct Trigger.
- The same analysis is valid for the case of the pseudostates. Because not all the DSLs of the family support the same pseudostates, we separate each of them in a different language module (i.e., **Fork**, **Join**, **DeepHistory**, **ShallowHistory**, **Junction**, **Choice**, **Conditional**). The core module only contains the initial pseudostate that included in all the DSLs of the family under study and that is required for expressing even a simple state machine. All the other pseudostates are considered as *extensions* to the core. Note that the final state is not a pseudostate. In the UML specification, it is consider as an special type of State (a subclass of state). As a result, final states are not included in this analysis.
- In order to support the semantic variation point regarding the execution order of the transition effects, we decide to separate this construct in a different language module (i.e., **TransitionEffect**). This decision permits to isolate the construct and its corresponding semantics in a different module that can be configured according to a given decision at the moment of the configuration of a DSL.
- A similar analysis is valid for supporting the semantic variation point about the transitions priority. Because this variation point is directly linked to the notion of composite states (i.e., if there are not composite state then there are not conflicting transitions so no priority should be defined), we encapsulate this semantics with the notion of composite states.

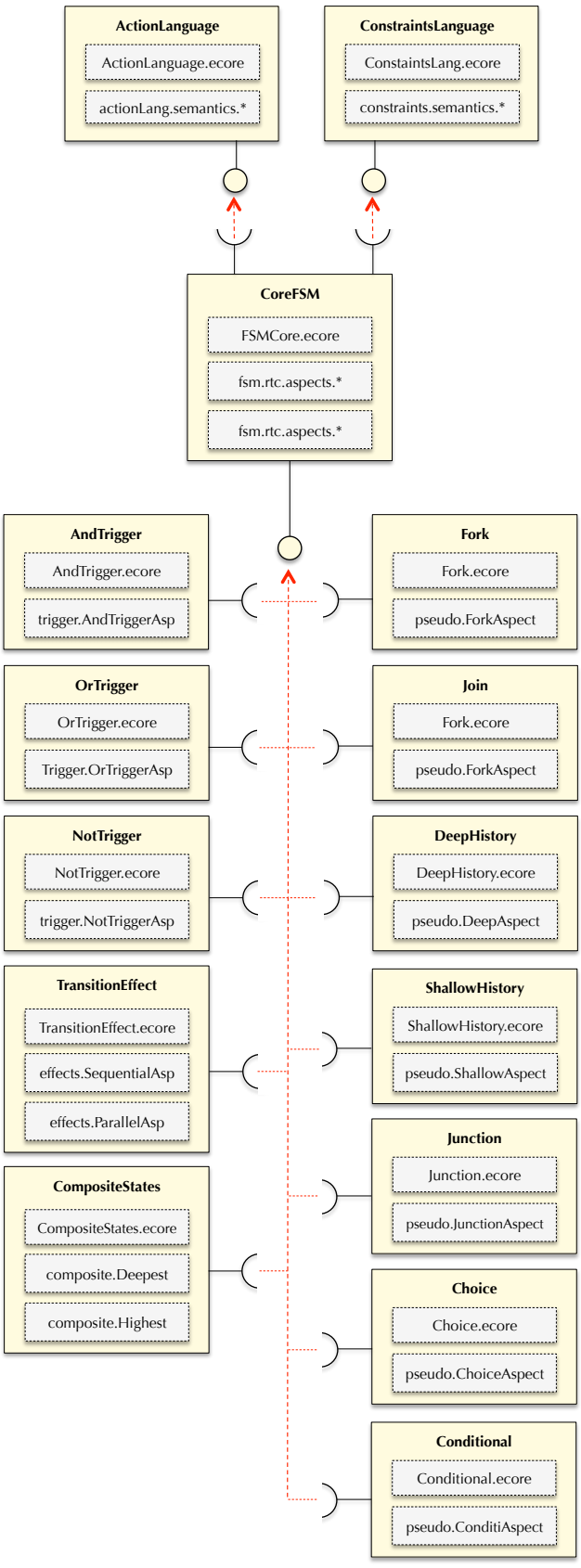

Fig. 18
A modular design for a family of DSLs for state machines

3.3. Variability management for family of DSLs for state machines

Figure 19 presents the variability model we build for the family of DSLs for state machines. It can be seen as a summary of the variation points that we introduce in section X. Besides, we include the configurations concerning each of the DSLs that compose the family that we use as starting point.

4. Tool demonstration

In this section we present the tool that implements our approach for building families of DSLs applied in the case study of the state machines. To do so, we first introduce a running example that can be used for visualizing some of the differences explained in the family in action. The execution of that running example is later illustrated by some videos.

4.1. Running example

In this section we aim to show how the semantic differences among the DSLs discussed in this document affect the execution of a complex state machine. Hence, we first introduce a state machine and then we execute it by using each of the DSLs. We show how, with the same input, the configuration of the state machine is different in each step for each DSL.

Fig 20 presents the state machine we will use to compare the DSLs. It is composed of two regions each of which has its own initial and final states. In the first region, we also have a composite state with a conditional pseudostate that takes a decision according to some information in the environment. In particular, it dispatches a transition either to the state S_4 or to the state S_5 depending on the value of the variable *y* defined in a the transition going from the state S_2 to the state S_3. Finally, the states finish in the state S_6.

In the region two we also have a composite state (S_8). In this case, the state introduces two conflictive transitions: the one going from the state S_9 to the state S_{10} and going from the state S_8 to the state S_11. These transitions are in conflict because they are dispatched in the same configuration (i.e., S_9, S_{10}) and they are associated to the same event (e_4).

4.1..1 Execution results

Table 4.1..1 presents the execution results for the complex state machine introduced above. The initial configuration of the state machine is given by the initial states. Then, it is the same for the three DSLs (S_1, S_7). However, from the first step the semantic differences start to appear. Let us discuss each case in detail.

Harel statecharts: In the **step 1** the machine receives the events $\mathbf{e}_0$ and $\mathbf{e}_3$. Because Harel's statecharts offers support for attending simultaneous events, then the two events are processed in the same step. The event e_0 fires the transition from S_1 to S_2 in the region at the top whereas the event e_3

Harel's state charts	
Step {**events**}	**Configuration**
step 0	[S_1, S_7]
step 1 {$\mathbf{e}_0$, $\mathbf{e}_3$}	[S_2, S_8, S_9]
step 2 {$\mathbf{e}_1$}	[S_3, S_4, S_8, S_9]
step 3 {$\mathbf{e}_2$}	[S_6, S_8, S_9]
step 4 {$\mathbf{e}_4$}	[S_6, S_{11}]
step 5 {$\mathbf{e}_5$}	[S_6, S_{12}]
-	-

Table 1
Execution trace of a Harel state chart

UML & Rhapsody	
Step {**events**}	**Configuration**
step 0	[S_1, S_7]
step 1 {$\mathbf{e}_0$, $\mathbf{e}_3$}	[S_2, S_7]
step 2 {$\mathbf{e}_0$, $\mathbf{e}_3$}	[S_2, S_8, S_9]
step 3 {$\mathbf{e}_1$}	[S_3, S_5, S_8, S_9]
step 4 {$\mathbf{e}_2$}	[S_6, S_8, S_9]
step 5 {$\mathbf{e}_4$}	[S_6, S_8, S_{10}]
step 6 {$\mathbf{e}_5$}	[S_6, S_{12}]

Table 2
Execution trace of a UML and Rhapsody state machines

Stateflow	
Step {**events**}	**Configuration**
step 0	[S_1, S_7]
step 1 {$\mathbf{e}_0$, $\mathbf{e}_3$}	[S_2, S_7]
step 2 {$\mathbf{e}_0$, $\mathbf{e}_3$}	[S_2, S_8, S_9]
step 3 {$\mathbf{e}_1$}	[S_3, S_5, S_8, S_9]
step 4 {$\mathbf{e}_2$}	[S_6, S_8, S_9]
step 5 {$\mathbf{e}_4$}	[S_6, S_8, S_{11}]
step 6 {$\mathbf{e}_5$}	[S_6, S_{12}]

Table 3
Execution trace of a Stateflow state machine

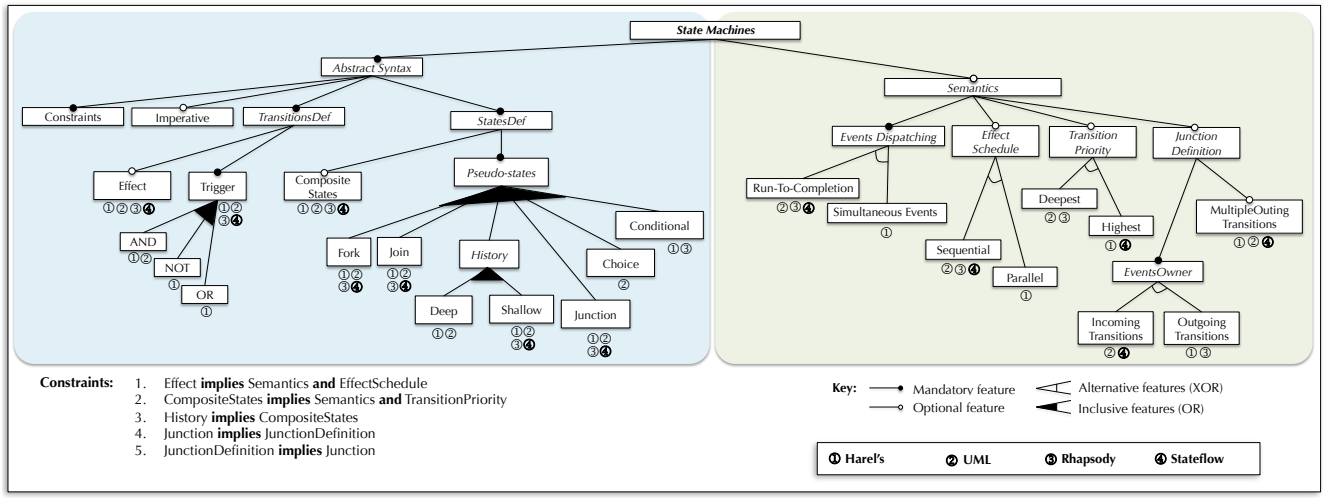

Fig. 19
Variability model of the family of state machines' languages

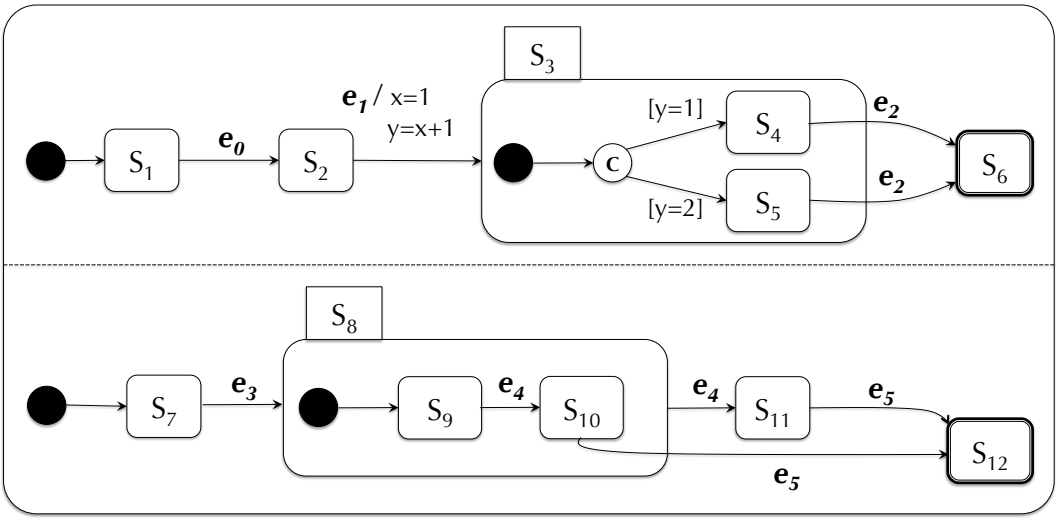

Fig. 20
Example of a state machine including several constructs

fires the transition from S_7 to S_8 in the region at the bottom. The state S_8 is a composite state that whose initial state references the state S_9. As a result, the configuration of the state machine after the first step is $\mathbf{S_2, S_8, S_9}$.

In the **step 2** the machine receives the event $\mathbf{e}_1$. This event fires the transition from S_2 to S_3 that executes two transition effects ($x = 1$ and $y = x + 1$). Because transition effects in Harel's statecharts are executed in parallel, the value of both x and y is 1. This has an impact on the execution of the state machine. In particular, the initial state of the composite state S_3 goes to a conditional pseudostate that goes either to S_4 or to S_5 depending on the value of y. In this case, the machine goes to the state S_4. As a result, the configuration after the execution of the event e_1 is $\mathbf{S_3, S_4, S_8, S_9}$. Note that in this step there are no events affecting the region at the bottom so it remains in the states S_8, S_9.

In the **step 3** the machine receives the event e_2. This event fires the transition going from the state S_4 to the sate S_6 while leaving the state S_3. The configuration after this step is $\mathbf{S_6, S_8, S_9}$. Similarly to the step 2, in this case the region at the bottom is not affected by the event.

In the **step 4** the event e_4 arrives to the state machine and produces a dispatching conflict. Since the current configuration of the machine is S_6, S_8, S_9 there are two transitions that can be fired: the one going from S_9 to S_{10} and the one going from S_8 to S_11. Because in Harel's statecharts the priority is given to the highest transition, then the later transition is fired. The configuration of the machine after the execution of the step is S_6, S_11.

Finally, in the **step 5** the the event $\mathbf{e}_5$ arrives and it fires

the transition from $S_1 1$ to $S_1 2$. After that, the configuration of the state machine is $\mathbf{S_6, S_1}2$. Because both states are final states the execution of the state machine finishes.

UML & Rhapsody: With the semantic variation points we present in this example, UML and the Rhapsody are equivalent. Then, they have the same execution results presented in the following.

In the **step 1** the state machines receives the events $\mathbf{e}_0$ and $\mathbf{e}_1$ that cannot be attended at the same time because neither UML nor Rhapsody support simultaneous events. Rather, they comply to the run to completion principle. Then, the machine processes first the event e_0 and then the event e_3. As aforementioned, the events are attended in the order in which they are expressed in the input but this is not necessarily always like this. The result of this first execution input, the state machine goes to the configuration $\mathbf{S_2, S_7}$ and in the **step 2** that is launched automatically the state machine goes to the configuration $\mathbf{S_2, S_8, S_9}$. Note that the configuration of the state machine is the same than the produced by Harel's statecharts with the difference that in this case the result is produced in two subsequent steps rather than in one.

In the **step 3** the state machine receives the event e_1 that executes two transition effects ($x = 1$ and $y = x + 1$). In this case, these effects are executed sequentially so the value of x is 1 whereas the value of y is 2. As a result, the conditional pseudostate in the state S_3 goes to the state S_5. The configuration of the machine after this step is $\mathbf{S_2, S_5, S_8, S_9}$.

In the **step 4** the machine receives the event $\mathbf{e}_2$ and the state machine fires the transition from the state S_5 to the state S_6. Then the configuration after the execution of the step is $\mathbf{S_6, S_8, S_9}$.

In the **step 5** the machine receives the event $\mathbf{e}_4$. In this case, the resulting conflict is resolved differently with respect to the case of Harel's statecharts. Concretely, in both UML and Rhapsody the priority on conflicting transitions is given to the one with the lower scope. That is, the transition located deepest in the hierarchy. As a result, the machine moves to the state S_{10} and the configuration after the step is $\mathbf{S_6, S_8, S_{10}}$.

Finally, in the **step 5** the the event $\mathbf{e}_5$ arrives and it fires the transition from $S_1 1$ to $S_1 2$. The execution of the state machine finishes.

Stateflow: The execution results of the state machine in Stateflow combines some the decisions made in the Harel's statecharts with the decisions made in UML & Rhapsody.

In the **step 1** the state machines receives the events $\mathbf{e}_0$ and $\mathbf{e}_1$ that cannot be attended at the same time because Stateflow does not support simultaneous events. Rather, they comply to the run to completion principle. Then, as the same as in the case of UML and Rhapsody the state machine first foes to the state $\mathbf{S_2}$ $\mathbf{S_7}$ and then to the state $\mathbf{S_2, S_8, S_9}$ in two steps.

In the **step 3** the state machine receives the event e_1 that executes two transition effects ($x = 1$ and $y = x + 1$). After that the value of x is 1 and the value of y is 2 because effects are executed sequentially. As the same as UML and Rhapsody, the configuration after the step 3 is $\mathbf{S_2, S_5, S_8, S_9}$.

In the **step 4** the machine receives the event $\mathbf{e}_2$ and the state machine fires the transition from the state S_5 to the state S_6. Then the configuration after the execution of the step is $\mathbf{S_6, S_8, S_9}$.

In the **step 5** the machine receives the event $\mathbf{e}_4$. In this case, the resulting conflict is resolved as the same as in the case of Harel's state charts so the configuration of the machine after this step is S_6, $S_1 1$.

Finally, in the **step 5** the the event $\mathbf{e}_5$ arrives and it fires the transition from $S_1 1$ to $S_1 2$. The execution of the state machine finishes.

4.2. Screen-cast and videos

After executing our tool, the user will find a screen as the presented in Fig. 21. There is a variability model written in CVL that permits to configure the variation points presented in this document. Then, that variability model is used for automatically generating a Melange script that is able to perform the composition of the language. Once a DSL is configured, we launch the state machine introduced in section 4.1. and we see how the out changes in each case.

The videos can be found in the following links:

- **Harel's statecharts:** `http://quick.as/mvprc1w4p`
- **UML & Rhapsody:** `http://quick.as/ZmbXIYePj`
- **Stateflow:** `http://quick.as/r92RUnqPZ`

5. Conclusions and perspectives

In this document we have reported on an engineering effort for providing a highly configurable language for state machines. The intention is to facilitate the adaptation of a state machine language to the specific modeling needs appearing during the construction of software intensive systems. There is, however, a long path to follow. In particular, we now aim to go to an abstraction process that allows us to understand the problem in the general context of software languages engineering and not only for the context of Thales and the state machines languages. To do so, we vision to tackle two main issues:

5.1. How to breakdown a language?

One of the main conclusions we obtain from this engineering exercise is that languages modularization os of vital importance for supporting languages variability. We need to be able to decompose a software language in several language modules that can be associated to the features expressed in a variability model. However, decomposing a language in several language modules is not only about offering a languages benchmark that enables modular definition of metamodels, grammars and semantics; it is just the technical part. The other important challenge is to understand how the language modules should be defined so they can be reused in other

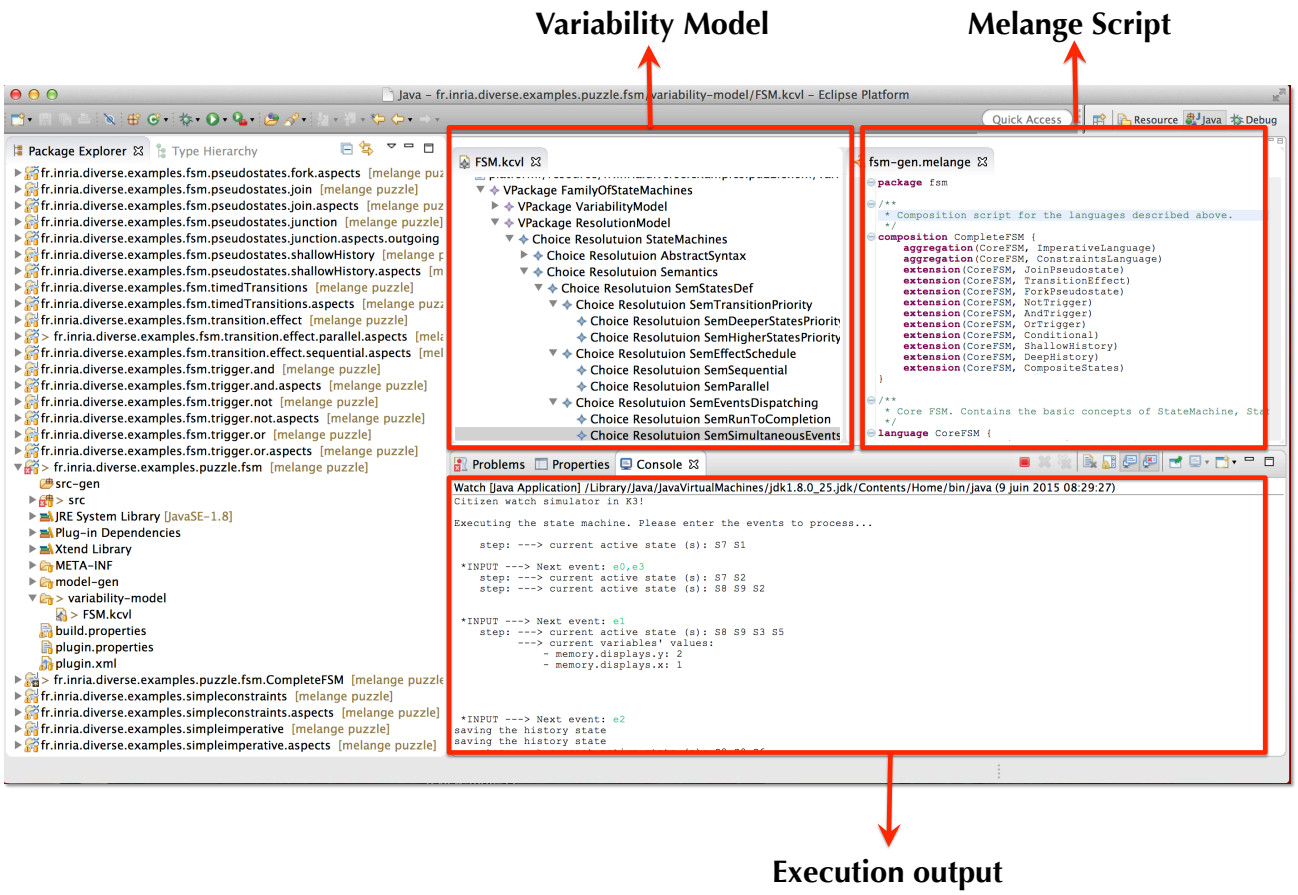

Fig. 21
Example of a state machine including several constructs

contexts. What is the correct level of granularity? What are the *"services"* that a language unit should offer for being considered reusable? What is the meaning of a *"service"* in the context of software languages? What is the meaning of a *"services composition"* in the context of software languages?

5.2. Support for multi-staged configuration

Another challenge to address when supporting variability in the construction of software languages is related with multi-staged languages configuration. That is, the possibility of configuring a language in several stages and by different stakeholders. This idea was introduced by Czarnecki et. al. [5] for the general case of software product lines and discussed by Dinkelaker et. al. [7] for the particular case of software languages. The main motivation for supporting such functionality is to transfer certain configuration decisions to the final user so he/she can adapt the language to exactly fits his/her needs [7].

References

[1] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[2] María Victoria Cengarle, Hans Grnniger, and Bernhard Rumpe. Variability within modeling language definitions. In Andy Schrr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 670–684. Springer Berlin Heidelberg, 2009.

[3] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.

[4] MichelleL. Crane and Juergen Dingel. Uml vs. classical vs. rhapsody statecharts: not all models are created equal. *Software & Systems Modeling*, 6(4):415–435, 2007.

[5] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In RobertL. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Berlin Heidelberg, 2004.

[6] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. Melange: a meta-language for modular and reusable development of dsls. In *Software Language Engineering*, Lecture Notes in Computer Science. Springer International Publishing, 2015.

[7] Tom Dinkelaker, Martin Monperrus, and Mira Mezini. Supporting variability with late semantic adaptations of domain-specific modeling languages. In *Proceedings of the First International Workshop on Composition and Variability co-located with AOSD'2010*, 2010.

[8] Holger Eichelberger and Klaus Schmid. A systematic analysis of textual variability modeling languages. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 12–21, New York, NY, USA, 2013. ACM.

[9] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications*, LDTA '12, pages 7:1–7:8, New York, NY, USA, 2012. ACM.

[10] Clément Guy, Benoît Combemale, Steven Derrien, Jim RH Steel, and Jean-Marc Jézéquel. On model subtyping. In *Modelling Foundations and Applications*, pages 400–415. Springer, 2012.

[11] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[12] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml). In Hartmut Ehrig, Werner Damm, Jrg Desel, Martin Groe-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkmper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2004.

[13] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[14] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, Oct 2004.

[15] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480. ACM, 2011.

[16] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and Franois Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[17] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[18] Holger Krahn, Bernhard Rumpe, and Steven Vlkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[19] Jörg Liebig, Rolf Daniel, and Sven Apel. Feature-oriented language families: A case study. In *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, VaMoS '13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[20] Roberto E. Lopez-Herrejon, Lukas Linsbauer, Jos A. Galindo, José A. Parejo, David Benavides, Sergio Segura, and Alexander Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353 – 369, 2015.

[21] Nadia Martaj and Mohand Mokhtari. Stateflow. In *MATLAB R2009, SIMULINK et STATEFLOW pour Ingénieurs, Chercheurs et Étudiants*, pages 513–586. Springer Berlin Heidelberg, 2010.

[22] B. Mora, F. Garca, F. Ruiz, and M. Piattini. Graphical versus textual software measurement modelling: an empirical study. *Software Quality Journal*, 19(1):201–233, 2011.

[23] Object Management Group (OMG). Uml 2.4.1 superstructure specification, 2011.

[24] Marko Rosenmüller, Norbert Siegmund, Thomas Thüm, and Gunter Saake. Multi-dimensional variability modeling. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 11–20, New York, NY, USA, 2011. ACM.

[25] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software & Systems Modeling*, 6(4):401–413, 2007.

[26] Edoardo Vacchi and Walter Cazzola. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures*, 43:1 – 40, 2015.

[27] Edoardo Vacchi, Walter Cazzola, Suresh Pillay, and Benot Combemale. Variability support in domain-specific language development. In Martin Erwig, RichardF. Paige, and Eric Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 76–95. Springer International Publishing, 2013.

[28] M. P Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

[29] J. White, James H. Hill, J. Gray, S. Tambe, A.S. Gokhale, and D.C. Schmidt. Improving domain-specific language reuse with software product line techniques. *Software, IEEE*, 26(4):47–53, July 2009.