A model-based approach to families of embedded domain specific languages

Jesús Sánchez Cuadrado and Jesús García Molina

Abstract— **With the emergence of model driven engineering (MDE), the creation of domain specific languages (DSL) is becoming a fundamental part of language engineering. The development cost of a DSL should be modest, compared to the cost of developing a general-purpose programming language. Reducing the implementation effort and providing reuse techniques are key aspects for DSL approaches to be really effective.**

In this paper we present an approach to build embedded domain specific languages applying the principles of model driven engineering. On the basis of this approach we will tackle reuse of DSLs by defining families of DSLs, addressing reuse both from the DSL developer and user point of views. A family of DSLs will be built up by composing several DSLs, so we will propose composition mechanisms for the abstract syntax, concrete syntax and model transformation levels of a DSL's definition. Finally, we contribute a software framework to support our approach, and we illustrate the paper with a case study to demonstrate its practical applicability.

Index Terms— **domain specific languages, model driven development, families of DSLs, DSL composition**

I. Introduction

DOMAIN specific languages (DSL) are becoming more and more important with the emergence of the model driven software engineering (MDE) paradigms, such as Model Driven Architecture (MDA), generative programming or software factories. DSLs allow programs to be written at an abstraction level closer to the problem domain than general-purpose programming languages. MDE technology includes model transformation languages intended to the generation of software artifacts (e.g. source code or configuration files) from specifications expressed by a DSL program.

The development of DSLs shares some similarities to the development of general-purpose programming languages. But, on the contrary to them, the cost of building a DSL should be modest. Several approaches have been proposed to define DSLs [1] [2] [3], and a number of studies report successful cases of DSL usage [4] [5] [6]. A well-known technique to create a DSL is embedding the DSL's constructs into an existing general purpose language, which acts as the host language. Such a DSL is therefore called an *embedded DSL*, and it inherits all features of the host language. This form of developing DSLs is widespread in the Haskell and Lisp functional language communities [7] [8], as well as in dynamic object oriented language communities, such as the Smalltalk and Ruby ones [9] [10].

An important advantage of the embedded approach is that it allows domain specific languages to be easily and rapidly developed [7] [11]. The implementation is usually straightforward since the language syntax, the type system, and the run-time system are only variations of the ones provided by the host language. However, the development of an embedded DSL tends to be an *ad-hoc* process, where developers just use host language idioms, neglecting the separation between concrete syntax, abstract syntax and semantics. On the other hand, MDE principles do promote this separation, and provide a conceptual framework to build DSLs in a systematic way [2] [3].

Defining a family of languages has been proposed as an approach to promote systematic reuse in the development of several DSLs for a given domain [12] [13]. A basic mechanism to achieve such reuse is the composition of DSLs. However, little attention has been paid to this topic, and DSLs are usually developed as standalone entities.

In the last three years we have developed a tool for model driven engineering, named AGE, which is based on DSLs embedded into the Ruby language [11]. The developer is provided with embedded DSLs for the different aspects in MDE, such as model transformation, code generation or validation and with a metamodeling facility compatible with Ecore [14].

This paper aims to present an approach to build embedded DSLs applying the principles of model driven engineering, that is, metamodeling and model transformations. On the basis of this approach, we will tackle the development of families of DSLs as a way of promoting reuse, both from the DSL developer and user point of views. Composition of DSLs will be a key point of our proposal. We will argue that supporting DSL composition implies providing composition mechanisms at all the levels of the definition of a DSL. We contribute a software framework that supports our approach.

This paper is organized as follows. The next section introduces the main concepts and motivates our proposal. Section III explains techniques to create embedded DSLs, while Section IV presents our framework integrating embedded DSLs with MDE, along with the composition mechanisms it provides. Section V addresses the problem of creating families of DSLs. In Section VI a discussion about the advantages and disadvantages of the embedded approach is presented. Some guidelines to decide when to use embedded DSLs are also given. Finally, Section VII presents the related work, and Section VIII gives the conclusions.

II. Overview

Domain specific languages are software languages tailored to address problems in some application domain. DSLs have a higher abstraction level than general-purpose languages since they provide constructs representing concepts of the domain. A DSL is esentially composed of three components [3] [6] [13], namely:

- *Abstract syntax*. The set of language concepts and their relationships, along with the rules to combine them.
- *Concrete syntax*. Defines the notation the end user will use to specify programs conforming to the abstract syntax. Textual and graphical notations are the most usual ones, but other representations such as tables can be defined as well.

This is a preliminary version. See the IEEE TSE site for the final version.
J. Sánchez and J. Molina are with the University of Murcia.

Preliminary version

- *Semantics*. Describes the meaning of the language's constructs. There are several approaches to semantics [3] [13], but their discussion is out of the scope of this paper. We will use translational semantics.

Model driven engineering provides a foundation to build DSLs by applying metamodeling. Metamodels provide a unified and expresive way to define the concepts of the domain. Object oriented metamodeling languages such as MOF [15] or Ecore [14] are typically used to define a language's abstract syntax. In the case of textual DSLs, formalisms such as grammars or XML schemas can be used to the define the concrete syntax. Then, a bridge between MDE and the chosen technical space (e.g. grammarware or XML) must be established. Finally, translational semantics fits well in an MDE approach, since the mapping between concepts of the source and the target language can be established using model transformations.

A *DSL definition* comprises the three components mentioned above. A *DSL program* is a piece of software, expressed using some concrete syntax, conforming to the DSL definition. In an MDE setting, programs are equivalent to models, so they can be manipulated using regular model operations (e.g. model transformations).

A *family of DSLs* is a set of related DSLs for a given domain. Each DSL is called a family member. A member may be linked to other members to address reuse either at the language definition level or at the user level. Developing a family of DSLs poses a challenge on how to compose its members.

An *embedded domain specific language* is a DSL which has been implemented on top of some general purpose language which serves as the host language. It reuses the infrastructure of the host language (e.g. concrete syntax, type system and runtime system) extending it with domain specific constructs. Thus, a DSL program expressed by an embedded DSL is a legal program of the host language.

The concrete techniques used to implement an embedded DSL depend on the paradigm the host language belongs to, because the abstractions provided by the host language are used to create domain specific constructs on top of them. Although embedded DSLs can be defined within any language, those languages providing a non-intrusive syntax are more suitable. Thus, functional and dynamic object oriented languages have been mainly used.

Several embedded DSLs have been defined for statically typed functional languages such as Haskell [7] [16], where lambda abstractions and monads are used to embed the DSL. In the case of Lisp-like dynamic functional languages the macro system is used [8]. In dynamic object oriented languages, such as Ruby or Smalltalk, the basic computation mechanism is message passing (i.e. method calls), while code blocks are first-class citizens. These two features have been used, for example in Ruby on Rails or Seaside [9], to define web frameworks based on embedded DSLs.

The embedded approach is a powerful way to build DSLs, however there is little or no separation between the language components, and its development tends to be rather ad-hoc. On the other hand, several approaches and tools intended to create DSLs relying on MDE techniques [2] [3] [17] has been proposed, however more research is necessary to address key issues such as the reuse of DSLs, where defining families of DSLs plays an important role. Our experience applying MDE techniques to embedded DSLs has shown that benefits on both sides can be achieved. On the embedded side the development follows a more systematic process where the different language components are clearly separated. The use of metamodels to define the abstract syntax helps reasoning about the domain and makes it possible interoperability with MDE tools. On the MDE side, taking advantage of the flexibility of embedded DSLs allows powerful tools to be created [11], which may help us to address research challenges, for instance reuse of DSLs.

We have combined MDE and embedded DSLs to tackle the development of families of DSLs. Each family member may be created with reuse or for reuse. We argue that composition mechanisms are needed at abstract syntax, concrete syntax and tranformation levels to achieve such reuse.

This paper reports our results using MDE and embedded DSLs in order to deal with families of DSLs. We have developed an MDE framework based on embedded DSLs to support our approach. It automatizes the development of DSLs and provides a number of composition mechanisms. The development of families of DSLs will be tackled relying on such composition mechanisms.

III. Embedded Domain Specific Languages

In this section we explain techniques to embed DSLs into dynamic object oriented languages, which are the technical basis for our proposal. We begin by introducing the example that will be used to illustrate the concepts and techniques discussed throughout this paper. We will use Ruby as the host language for embedding the DSLs.

A. Running example

This example tackles a problem related to task automation in the development of Eclipse contributions [18]. Developing a contribution to the Eclipse platform involves filling configuration files and writing repetitive code, which refers to elements defined by the configuration files in no obvious ways. In the end, the knowledge about how to develop a contribution (e.g. adding a button with a shortcut, or defining a new wizard) is scattered through several classes and configuration files, yielding to a copy-paste reuse process from existing implementations.

The Eclipse platform is very large, so we will explain only a few concepts needed to understand the example. Eclipse provides the developer with extension points to specify contributions and to extend the platform. Different types of projects can be defined, for instance to support several programming languages. Resources, such as files, can be classified in resource types to associate specific actions, source editors, etc. to them. Finally, to allow users to invoke actions, elements such as buttons, pop-up menus, or key shortcuts are added to the user interface[1].

In order to automatize the development of Eclipse contributions, a family of DSLs can be defined, where each member in the family allows related Eclipse contributions to be specified. Also, this family provides a higher level of abstraction than the traditional way of developing for the Eclipse platform.

Given a set of DSL programs belonging to the family, a process consisting of a chain of model-to-model and model-to-code transformations automatically generates most of the contribution code. This paper will be illustrated with three DSLs of this family, called *Resources*, *Actions* and *Modeling Actions*.

[1]For the sake of simplicity, in this explanation we do not introduce core Eclipse concepts such as plugin, perspective or view.

Preliminary Version

Fig. 1. Metamodel capturing concepts related to resources, projects and actions, and their relationships.

- The *Resources* DSL allows us to describe “physical” resources such as types of projects, the kinds of resources (e.g. a kind of file) a project may hold, etc.
- The *Actions* DSL allows us to define actions a user can perform, as well as specifying the widget used to activate the action (e.g. a button, a key shortcut, etc.). As we will see, it “references” the Resources DSL to associate actions with resource types.
- The *Modeling Actions* DSL is an extension of the Actions DSL, which provide actions specific to an MDE environment, such as launching a transformation or validating a model.

The following piece of code shows the concrete syntax of the Resources DSL. In particular, it shows an excerpt of a program to define a resource type for transformation files, and a project type for our MDE tool, AGE. This DSL has been defined on the basis of the metamodel shown in Figure 1.

```
resource_type 'rubytl-transf' do
  extension '.rtl'
  icon 'rubytl.gif'
end

project 'age' do
  folder 'models'
  folder 'metamodels'
  folder 'transformations' do
    contains_res 'rubytl-transf'
  end
end
```

First, a resource type is defined for RubyTL transformation files (i.e. we have created a model transformation language named RubyTL). It has an extension and an icon associated. Then, a new type of project is defined, which is composed by several folders to store artifacts such as models, metamodels or transformation definitions. Also, it specifies that the `transformations` folder contains `rubytl-transf` resources.

As is shown by the metamodel of Figure 1, an action group contains one or more action definitions. An action can be related to a certain resource type to make this action applicable only to this kind of resources, while an action group is related to a certain kind of project. The example below corresponds to the Actions DSL. An action group for the `age` project type is defined, which contains only one action, related to RubyTL transformation resources. The action will be shown in a pop-up menu.

```
action_group 'age-group' do
 project 'age'

 action 'export transformation' do
    resource 'rubytl-transf'
    show_in_popup
 end
end
```

It is important to notice that the two code excerpts shown above correspond to two different DSL programs. However, action groups and actions defined in the second program need to reference resource types defined in the first one. This topic will be addressed in Section V. Also, in such a section, the Modeling Actions DSL will be defined as an extension of Actions.

B. Basic implementation technique

The basic strategy is to create an embedded DSL on the basis of the keywords it offers, and on how they can be composed. Usually, a keyword will represent a domain concept, and keyword composition will determine the relationships between the concepts. As we have pointed out, depending on the paradigm the host language belongs to, and the abstractions it provides, the constructs used to embed the DSL are different. Since we are dealing with dynamic object oriented languages, each DSL keyword will be defined as a method, with zero or more parameters, while nested structures will be mapped to a code block. In the simplest approach such methods are defined in some class where the whole DSL functionality is implemented.

As an example, the implementation of the `project` and `folder` keywords could be as follows. A method is defined for each keyword, and the method's implementation creates a data structure to be processed subsequently. Implementation-wise a new `ProjectType` object is added to a list of project types when the `project` method is executed, then the passed block is evaluated using `yield`. The `folder` method adds a `Folder` object to the list of folders maintained by its enclosing project type.

```
def project(name, &block)
  @projects << ProjectType.new(name)
  yield     # synonym for "block.call"
end

def folder(name, &block)
  folder = Folder.new(name)
  current_project = @projects.last
  current_project.folders << folder
  yield if block_given?  # the block is optional
end
```

We propose to use metamodels to define the abstract syntax of the embedded DSL, so that keyword implementation is in charge of creating an instance of the abstract syntax, instead of creating an arbitrary data structure. Thus, when the DSL program is evaluated the result is a model conforming to the abstract syntax, which can be manipulated in a number of ways: serialized to some interchange format (e.g. XMI), transformed using model transformation languages or just interpreted. Embedded DSL techniques are thus used to give concrete syntax to the abstract syntax metamodel of a DSL.

We provide the following guidelines to establish the mapping between an abstract syntax metamodel and those constructs of DSLs embedded into dynamic object oriented languages.

Each *metaclass* is mapped to a DSL keyword, implemented by a method. Some attributes of the metaclass can be specified as parameters of the keyword, typically compulsory attributes such as a name for the concept. In the example, the `ProjectType` metaclass is mapped to the `project` method.

A *containment relationship* is mapped to a code block, where keywords representing the referenced metaclass are written inside the block. The container is specified as the outermost DSL construct, while the contained elements are specified inside the

Preliminary version

block. An element is owned only by one container, so the relationship is univocal, and there is no need to reference the container explicitly. In the example it is enough to enclose folders between `do - end` within the project they are related to, because a folder is contained in only one project type

A *non-containment relationship* implies that the element to be referenced is defined in some place in the DSL program. Some kind of identifier has to be used to reference such an element. A non-containment relationship can therefore be mapped to a keyword enclosed within a code block, as in containment relationships, but the keyword receives an identifier representing the referenced concept as a parameter. In the example, a `folder` references the resource it contains by means of the `contains_res` keyword. The DSL implementation must look up the resource definitions and connect the corresponding one to the folder.

Optional attributes, attributes with default values, and in general attributes not specified as parameters, can be specified in the enclosing code block using a keyword. For instance, the `iconPath` attribute is specified by the `icon` keyword.

These are basic guidelines, but variations may be considered. For example, a containment relationship can be represented in the concrete syntax as a non-containment relationship, but it is more clear to use the strategy explained above, since it avoids the need of using identifiers to reference elements.

It is also possible to take advantage of features of the host language to make the syntax more compact or readable. For instance, declaring a folder can be shortened if Ruby hashes (i.e. name-value pairs specified by `=>`) are considered.

```
folder 'transformation',
        :contains_res => 'rubytl-transf'
```

To sum up, we propose to use those constructs and techniques of embedded DSLs to give concrete syntax to an abstract syntax metamodel. The rest of the section is devoted to describe techniques intended to improve quality attributes of embedded DSLs such as expresivity, reusability and usability.

C. Leveraging expressiveness

The implementation technique explained until now only allows us to define block-structured DSLs. Nevertheless, several techniques can be used to leverage the expressiveness.

The operators provided by the host language can be used to define small expression languages on the DSL concepts. The host language must allow operator overloading, so that it is possible to use DSL constructs as if they were primitive constructs. For instance, one can create a facility to define a new type of project on the basis of an existing one, overloading the addition and substraction operators to add or remove project folders.

```
project_variant 'simple_variant' do
  project('rubytl') - folder('transformation')
                    + folder('code-generator')
end
```

In addition, some pure object oriented languages allow primitive types to be extended, so that new methods and operators can be added, or even existing operators can be overridden. Thus, domain specific operations can be added to primitive types. For instance, a new method can be added to the *String* class to implicitly perform operations on images. Below, an icon in gif format is transformed to a png image.

```
icon 'rubytl.gif'.as_png
```

A general approach to write rich expressions in an object oriented language is to chain expressions in cascade using the *dot notation* [19]. Each method call will return an object that allows other method calls to be chained. This approach is the only one available in object oriented languages such as Java, because they have an intrusive syntax and do not support closures or code blocks. For instance, a project variant could be defined from a base project by including and excluding folders.

```
project_variant('my_variant).from_base('rubytl').
   excluding('transformations).
   including('code-generation')
```

D. Restricting the host language

In the embedded approach, any syntactically correct program of the host language may be a legal DSL program. As far as possible, the capabilities of the host language should be restricted to ensure certain syntactic correctness from the DSL point of view.

Thus, one important concern is whether it is possible to restrict the availability of a given keyword to a certain scope. Usually, DSL keywords are defined as global methods or functions which can be used in any part of the DSL, and it is up to the DSL user not to write a keyword in a wrong place. For instance, if such a constraint cannot be enforced, the `folder` keyword can be written outside a `project` construct.

Dynamic languages supporting code blocks usually allow a code block to be evaluated in an execution environment different from the default one. We take advantage of this feature to create a "sandbox" (or namespace) for each set of DSL keywords that can be enclosed within another DSL keyword. The code block passed to such a keyword will be evaluated in its sandbox to ensure that only keywords defined in it can be used.

A *sandbox* is simply defined as a class with methods representing keywords. The following excerpt represents the sandbox for the `project` keyword.

```
class ProjectTypeSandbox
  def folder( name )
    ...
  end
end
```

In this way, when a code block is passed to the `project` keyword to specify elements related to a project (i.e. folder definitions), it is evaluated in the context of the project type sandbox. This means that a new instance of the sandbox is created, and the execution environment of the code block will be such an instance, so ensuring that only those methods defined in the sandbox are available.

The following excerpt shows the definition of the `project` keyword following this strategy. It expects the project type name, and a code block. The passed block is evaluated in the context of a new instance of the sandbox, using the Ruby built-in `instance_eval` method. It is worth noting the difference between the implementation shown in Section III-B and this one. In the former, we used *yield* which evaluates the passed block in its default environment, while now the block is evaluated in a more restricted environment.

```
def project(name, &block)
  @projects << ProjectType.new(name)
  sandbox = ProjectTypeSandbox.new
  sandbox.instance_eval(&block)
end
```

Preliminary Version

With this approach a *root* sandbox must be created to define the DSL's root keywords, that is, those keywords that are not enclosed within any other keywords (e.g. `project` is an example of root keyword). The main advantage of this technique is that several DSL programs can be evaluated, even in parallel, without conflicts between each other, because each one is evaluated within its own root sandbox instance.

However, another form of sandboxing is needed when primitive types are extended. Since each DSL may extend primitive types in its own way, when two or more embedded DSLs are executed in the context of the same virtual machine, name clashes can arise. To solve this problem, a form of sandboxing at virtual machine level must be provided by the host language, so that the modifications to primitive types made by one DSL do not have visibility outside their sandbox.

E. Keyword extension

The usage of sandboxes has an additional advantage related to the reuse of the concrete syntax of a DSL. Any sandbox can be independently extended using class inheritance, so that a new keyword can be defined from an existing one. This feature also means that a new DSL can be created from an existing one, just extending several keywords, and probably the root sandbox.

When a keyword K_{base}, with sandbox S_{base}, is extended to define a new keyword K_{ext}, the strategy is as follows.

- Creating a new sandbox S_{ext} as a subclass of S_{base}, defining within it keywords for new attributes and relationships.
- Identifying the keyword K_{parent} where K_{base} is contained.
- Creating a subclass of the corresponding sandbox for K_{parent}, either to create a new keyword or just overriding the keyword method for K_{parent} in order to instantiate the S_{ext} sandbox instead of S_{base}.

For example, in the Modeling Actions DSL a *transformation action* is a specialized kind of action, so it can be based on the existing *action* keyword defined in the Actions DSL. As shown below, creating the `transform_action` keyword on the basis on an `action` keyword, implies inheriting from the `action` keyword's sandbox to include new keyword methods, such as `transformation`. Also, the `transform_action` method is created in a class inheriting from the action group sandbox, because an action is contained into an action group.

```
class TransformActionSandbox < ActionSandbox
  def transformation(name)
    ...
  end
end

class ModelingGroupSandbox < ActionGroupSandbox
  def transform_action(name, &block)
    sandbox = TransformActionSandbox.new
    sandbox.instance_eval(&block)
  end
  ...
end
```

To complete the extension, the definition of the `action_group` keyword method must be overridden (i.e. inheriting from the corresponding sandbox, in this case the root sandbox) to instantiate the `ModelingGroupSandbox` instead of `ActionGroupSandbox`.

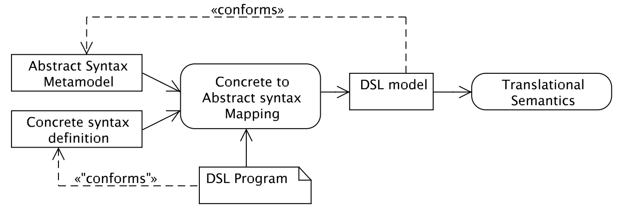

Fig. 2. Steps and artifacts involved in the definition of a DSL using our framework for embedded DSLs.

IV. A FRAMEWORK FOR EMBEDDED DSLS AND MODEL DRIVEN ENGINEERING

In this section we present our approach for developing DSLs, which integrates embedded DSLs with model driven engineering. Our proposal establishes a systematic process to clearly separate the three components of a DSL's definition.

Figure 2 illustrates the steps and artifacts involved in the definition of a DSL. The DSL's concrete syntax is defined relying on some formalism or technique, in our case the embedded approach. The DSL's abstract syntax is defined by a metamodel. . The concrete syntax to abstract syntax mapping establishes the correspondences between concrete syntax elements and metamodel elements. When a DSL program is evaluated, a model conforming to the abstract syntax metamodel is created[2] according to the mapping. Finally, we consider translational semantics to some target language or platform, which is achieved by a translation process involving model-to-model and model-to-code transformations, which eventually generates executable code.

We have developed a software framework to support this process. It is composed of several embedded DSLs and a metamodeling kernel compatible with Ecore (via XMI files following the EMF format to interoperate with Eclipse-based tools). A DSL which automatizes and simplifies the techniques explained in Section III to create concrete syntaxes embedded into Ruby has been created. It is complemented by another DSL to establish the mapping between a concrete syntax and an abstract syntax metamodel. Finally, we have developed a model-to-model transformation language called RubyTL [20].

With this framework, a DSL can be given a concrete syntax very easily taking advantage of the flexibility of the embedded approach. At the same time the use of metamodels to describe the DSL's abstract syntax helps reasoning about the domain, and promotes model transformations. These features allow rapid prototyping of DSLs, and we have in turn relied on them to create the embedded DSLs provided by the framework.

A distinctive feature of our framework is that each component provides composition mechanisms to achieve reuse at the corresponding level of the definition of a DSL. We will use these mechanisms in Section V to address the creation of families of DSLs. In the rest of the section we present each component of the framework, focusing on the composition primitives it offers.

A. Abstract syntax

The set of language concepts and their relationships, along with the rules to combine them, form the DSL's abstract syntax. The

[2]Notice that the *conforms* relationship between the DSL program and the concrete syntax definition has been quoted in Figure 2 to reflect the fact that, at the text level, the only conformance relationship that can be enforced is between the DSL program and the host language's syntax.

Preliminary Version

definition of abstract syntaxes using object oriented metamodeling is a well-known topic [3], and we will not focus on it, but on how to modularize a metamodel. We implement two mechanisms based on the notions of package import and package merge proposed by the UML/MOF specification [21]. The former allows us to establish dependencies between independent metamodels, whereas the later provides us with a means to create composite metamodels from existing ones.

When a metamodel imports another metamodel, the metaclasses of the imported metamodel are visible within it. This allows dependencies to be represented while still keeping the metamodels independent. In our tooling the dependencies are implemented as cross-references between metamodels.

Package merge is defined as a directed relationship between two packages, where the contents of a source package (*receiving* package) are extended with the contents of a target package (*merged* package). Matching elements are merged (e.g. two metaclasses with the same name), while non-matching elements are just deep-copied. A typical approach to metamodel modularization is to define a base metamodel which is later augmented by merging extensions defined in other metamodels.

Although the basic definition of package merge is conceptually simple, there are some concerns which make it difficult to understand and increases the complexity of the implementation [22]. For instance, the same metaclass may be partially defined in several metamodels, and conflicts may appear when merging attributes. In [13] a merge approach is used to handle metamodel variability. It allows us to leave the connections or joint points between metamodels open by defining "labels". When metamodels are merged the labels are substituted by the actual names of the metaclasses to be merged. The main problem of this approach is that the notion of label has to be propagated to any tool or formalism dealing with concrete syntax and semantics (model transformations in our case).

In this way, our strategy to merge metamodels is a simplified version of the two former approaches. It removes part of the package merge complexity by avoiding partial definition of elements (e.g. metaclasses), and there is no need to adapt the rest of the framework, because instead of using "labels" we just define a joint point as an empty metaclass with a name. The merging process can be briefly summarized as follows. Two metaclasses are merged when both names are identical. An error is raised if none of the matched metaclasses are empty (i.e. at least one of them must be a joint point). Subpackages are merged when their names match, merging their metaclasses as explained. The rest of the metamodel elements are just copied, arranging the relationships to the merged elements appropriately. This has been implemented as a model-to-model transformation using the RubyTL transformation language,

It is worth noting the difference between import and merge with regard to how they affect to the conformance relationship between a DSL program (i.e. a model) and its abstract syntax metamodel. With a merging strategy the actual metamodel to which a DSL program conforms to is the composite metamodel resulting from the merge. On the contrary, when a metamodel imports a metaclass from another metamodel, cross-references between models conforming to these metamodels will appear. We will use these mechanisms in Section V to address two different kinds of reuse.

B. Concrete syntax

As aforementioned, to define the concrete syntax of a metamodel we rely on the technique of embedding the DSL's syntax into the Ruby language. Usually, the definition of an embedded DSL repeats the same idioms explained in Section III, such as defining a method for each keyword, sandboxing, etc. Thus, we have developed a DSL intended to automatize this task.

Our DSL relies on the idea of keyword and keyword composition as the main abstractions to define the structure of an embedded DSL. The generation of the actual implementation is done dynamically, and transparently to the user. This generated implementation also takes into account error reporting, manipulating the exception trace to translate error messages automatically (e.g. to identify the error line). Three basic constructs are available:

- *Keyword*. This is the basic construct to create the DSL. The number and the type of the parameters are specified.
- *Keyword extension*. A keyword can be defined as an extension of another keyword.
- *Keyword composition*. This construct specifies which keywords can be nested within another keyword, including restrictions about the cardinality of the nested keywords.

The notions of keyword, keyword extension, and keyword composition hide the complexity of creating a sandbox for each context where a keyword can appear, as well as creating subclasses to define extended sandboxes. When defining the concrete syntax, the developer only needs to focus on the language keywords and their composition. Also, this DSL provides other features such as abstract keywords, or a means to define rich expression languages based on the techniques explained in III-C, but they are out of the scope of this paper.

For instance, an excerpt of the definition of the Resources DSL is shown below. The `project` keyword expects the project name of string type, as well as an optional parameter to allow a description to be specified The `resource_type`, `folder` and `contain_res` keywords are defined in the same way. The `composition_for` construct is used to specify which keywords must be enclosed within another one. In this case, `folder` and `contain_res` keywords are allowed only within a `project` and a `folder` keyword respectively.

```
keyword 'project' do
  params 'name', :string
  params 'description', :string, :optional
end
keyword 'folder' do
  params 'name', :string
end
keyword 'contain_res' do
  params 'resource_type', :string
end
keyword 'resource_type' do
  param :name, :string
end

composition_for 'project' do
  nested 'folder', :one_or_many
end
composition_for 'folder' do
  nested 'contain_res', :one
end
```

Once defined the concrete syntax of a DSL, the next step is to establish the mapping to the underlying abstract syntax.

Preliminary version

C. Concrete syntax to abstract syntax mapping

A bridge from embedded DSL constructs to the MDE technical space (e.g. metamodeling concepts such as metaclasses, relationships, etc.) must be created, so that the evaluation of a DSL program yields to the creation of a model. The mapping from the concrete syntax to the abstract syntax establishes such a bridge. In our experience, the nature of this mapping usually implies four kinds of operations:

- *Straightforward mappings*, where a keyword is directly mapped to a metaclass, and each keyword parameter corresponds to an attribute of the metaclass.
- *Establishing relationships* between metamodel elements already created by straightforward mappings. They can be either containment or non-containment relationships.
- *Global-to-local transformations* [23], where information to create a single target element is spread through several places of the source model (in this case, the concrete syntax model). Complex queries may be needed to retrieve the required information.
- *Initialization sentences*. Depending on the structure of the target abstract syntax metamodel, the creation of a given element may imply creating some others target elements imperatively to complete the mapped one.

An important part of the mapping is related to resolve identifier-based references at the concrete syntax level to explicit model references (i.e. a concrete syntax tree is converted to an abstract syntax graph). When an identifier is used to allow a concept to be referenced, it can take two forms: global or local.

- *Global identifiers*. A global identifier is unique between all the instances of a given metaclass. Therefore, once an instance has been given an identifier, no other instance of the same metaclass can have the same identifier.
- *Local identifiers*. A local identifier is unique only within a given scope. Therefore, it must be qualified with the identifier of its scope.

This issue has an impact in the way a metaclass is referenced. Global identifiers can be referenced without being qualified with the path to reach the identifier scope. On the contrary, to refer to a local identifier the complete path to reach the identifier scope must be given.

It is worth noting that, in an embedded DSL, identifiers need to be quoted (i.e. quotes are the way to specify a string). Otherwise the compiler would consider the identifier as an undefined variable or method, so provoking an error. In dynamic languages it is possible to use some metaprogramming strategies to avoid quoting identifiers, but they are out of the scope of this paper.

To automatize the task of defining the mapping between the concrete and the abstract syntax we have also created an embedded DSL which provides declarative constructs for this purpose. Also, to give more flexibility, a visitor [24] is in charge of traversing the syntax tree, allowing methods to be written in order to complete the mappings with queries and initialization sentences.

Mappings are established by three constructs: (1) the *map* construct establishes the correspondence between keywords and metaclasses (it is also in charge of mapping parameters to attributes), (2) the *con* construct is in charge of mapping keyword composition to containment relationships, while (3) the *ref* construct maps identifier-based relationships to explicit references.

The following piece of code establishes the mapping between the concrete syntax of the Resources DSL and its abstract syntax. Each keyword is mapped to the corresponding metaclass using the *map* construct (e.g. each ocurrence of `project` will lead to the creation of a `ProjectType` element). It is also used to map keyword parameters to attributes, and to establish whether an attribute acts as an identifier, being global or local. The containment relationship between a project and its folders is mapped using *con*. Regarding non-containment relationships, the *ref* construct uses the information about global or local identifiers declared with *map* to resolve the references.

```
mappings do
 map 'resource_type' => ResDSL::ResourceType
 map 'resource_type.name' => 'name', :id => :global
 map 'project' => ResDSL::ProjectType
 ...
 map 'folder' => ResDSL::Folder
 map 'folder.name' => 'name', :id => :local

 con 'project.folder' => 'folders'
 ref 'folder.contain_res' => 'resource_type'
end
```

When the concrete syntax to abstract syntax mapping is performed on a given source DSL program, the result is a model conforming to the abstract syntax metamodel. This model can be now manipulated using, for instance, model transformations.

D. Translational semantics

Model-to-model transformations play a key role to establish mappings between metamodels, and to convert high-level, abstract models to low-level, concrete programming languages and platforms. In the DSL setting, model-to-model transformations can be used to establish the translational semantics of a DSL by describing a mapping to another language. This is the approach taken in this paper. As part of our research in model transformations we have developed a model transformation language, named RubyTL [20], which has been implemented using the techniques explained in this paper.

RubyTL is a hybrid rule-based model transformation language, intended to specify mappings between metamodels. It has a declarative part based on rules and bindings. Bindings are a special kind of assignment which allow us to describe *what is transformed into what*, while rules are in charge of resolving them (i.e. a rule is implicitly called to resolve a binding). The imperative part of the language is given by the fact that RubyTL is embedded within Ruby. In this way, it is possible to write arbitrary Ruby code in the mapping part of a rule.

The following piece of transformation definition establishes the mapping between the Resources DSL metamodel and the Eclipse architecture metamodel shown in Figure 3. The first rule of the transformation definition creates a new `FileTypeEP` extension point from each `ResourceType` element. Equally, the second rule creates a `FileResource` element from each `ResourceType`. The only binding in the first rule establishes that the `resource` source element must be transformed to a `FileResource` and assigned to the `fileResource` property. This binding is resolved by the second rule.

```
transformation 'res2ecl'

rule 'resource2ep' do
  from ResDSL::ResourceType
  to   Arch::FileTypeEP
```

Preliminary Version

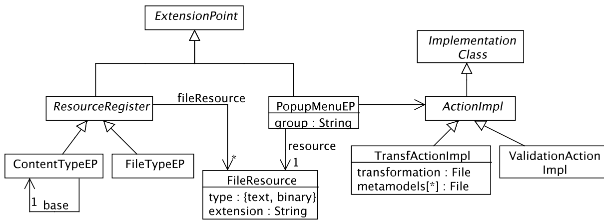

Fig. 3. Excerpt of a metamodel to describe an architectural solution to develop contributions for the Eclipse platform. Extension points are represented as metaclasses inheriting from the `ExtensionPoint` abstract metaclass, while already existing implementation classes are represented by metaclasses inheriting from `ImplementationClass`.

```
  mapping do |resource, filetype_ep|
    filetype_ep.fileResource = resource
  end
end

rule 'resource2file_resource' do
  from ResDSL::ResourceType
  to   Arch::FileResource
  mapping do |resource, file_resource|
    file_resource.extension = resource.extension
    file_resource.type      = Arch::FileModes::Text
  end
end
```

An important feature of RubyTL is that it provides a modularity mechanism, called phasing [25]. Unlike rule-level modularity mechanisms, such as the ones discussed in [26], phasing is coarser-grained, and it is intended to reuse and compose complete transformation definitions, not only individual rules.

At this point, we briefly explain our phasing mechanism, although a more complete description can be found in [25]. With a phasing mechanism, a transformation definition is organized as a set of phases, which are composed of rules. Executing a transformation definition consists of executing its phases in a certain order. The execution of a phase means executing its rules as if they belonged to an isolated transformation definition, without conflicts with rules defined in other phases. A transformation definition is therefore seen as a phase, so allowing the same composition operators as for phases.

We have defined two operators for composing phases. These operators are based on the trace information recorded during the execution of rules. A trace establishes that a certain source element has been transformed to a target element. We explain both operators assuming that the piece of trace model shown in Figure 4 has been recorded during the execution of the transformation shown above.

- *trace query*. It is a function which takes a source element as input and returns one or more target elements that are related by the trace to the source element (i.e. those target elements that have been created from the source element). The returned elements can be constrained to be instances of some metaclass. For instance, *trace_query(e1, FileResource)* returns { `f1` } because $e1$ is related to only one element of type `FileResource` by the trace.
- *refinement rule*. It is a special kind of rule which matches against the trace information, instead of the source model. There is a match if a source instance of the metaclass specified in the rule's source pattern (i.e. rule's *from* part), has a trace relationship with one target instance of the metaclass specified in the rule's target pattern (i.e. rule's *to* part). For each match the refinement rule is executed, but instead of creating a new target element as usual, the element matched by the target pattern is used. This means that no new target elements are created, but the rule works on existing elements, refining them. For example, the following rule would match any `FileResource` and `FileTypeEP` instances which are both related to the same source instance of type `ResourceType`. According to the trace model, the rule application will result in two matches: {`r1, e1, f1`} and {`r2, e2, f2`}. Some code could be written in the mapping part of the rule to refine these already existing target elements, {`e1, f1`} and {`e2, f2`} in this case.

```
refinement_rule 'resource_type' do
  from ResDSL::ResourceType
  to   Arch::FileTypeEP, Arch::FileResource
  mapping ...
end
```

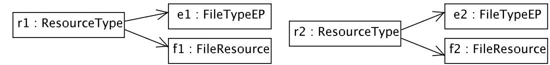

Fig. 4. Piece of a trace model recorded during a transformation execution. `r1` and `r2` are source elements, while `e1`, `e2`, `f1` and `f2` are target elements created by transformation rules. Arrows represent trace relationships.

Thus, our proposal for transformation composition relies on querying the piece of target model created by a previous transformation (or phase), using the trace information, either by applying the *trace query* or *refinement rule* mechanisms.

Finally, a transformation definition can be imported within another transformation definition, and it is treated as a regular phase. A construct called `scheduling` allows us to set the execution order of phases and imported transformation definitions. A practical application of these features is given in the next section.

V. Families of DSLs

Defining families of languages has been proposed as an approach to promote systematic reuse in the development of DSLs for a given context [12] [13]. A family of DSLs is defined as a set of related DSLs intended to address some task or problem in a given domain.

Practical development and usage of a family of DSLs requires reuse both from the DSL developer and user points of view. Some DSLs in a family may have commonalities, which can be used to factorize common DSL implementation code that is later composed with the corresponding variants. In the running example, the Modeling Actions DSL reuses most of the infrastructure of the Actions DSL. On the other hand, the user must be provided with modularization mechanisms to organize and reuse DSL programs. For example, the Actions DSL (or the Modeling Actions alternatively) and the Resources DSL provides the user with a means to reuse the corresponding DSL programs in different contexts (e.g. the same Resources program can be used for several projects).

To tackle these two forms of reuse, mechanisms for DSL composition are needed. DSL composition can be defined as the ability to relate two or more DSLs in order to achieve a certain functionality which is the result of combining the functionality of the composed DSLs. Thus, a single DSL can be decomposed into several, smaller DSLs than can be reused in different contexts, either by the developer or the user. In this way, we distinguish

Preliminary Version

between two forms of DSL composition: language composition and program composition.

- *Language composition* is the ability to compose two or more DSLs transparently to the user. This means that, from the user point of view, there is only one DSL with a certain functionality. This form of reuse allows the developer to take advantage of existing DSL implementations to create a new specialized DSL.
- *Program composition*, on the other hand, refers to the possibility of the user applying mechanisms to modularize the DSL programs he or she writes. This form of composition allows DSL programs to be reused.

Language composition requires the existence of modularization mechanisms at the tooling level, that is, mechanisms to modularize the definition of the concrete syntax, abstract syntax, and semantics are needed. On the contrary, program composition implies that the DSL developer provides the DSL user with modularization mechanisms intended to organize DSL programs.

There are several concrete forms of composition that can be classified into one of these two categories. In this paper we will illustrate language composition with *extension* and program composition with *importation*. Other forms of composition that are out of the scope of this paper are *merging* [13] and *superimposition* [27]. Merging consists of combining pieces of DSLs, establishing connection points, to form a composite DSL. Superimposition allows the user to superimpose additional functionality to an existing DSL program.

A. Approach overview

Each member of a DSL family must be developed as a reusable unit, which must be composed with other members of the family to provide a common functionality. Typically, a member corresponds to some aspect or concern of the domain of interest. A member has a corresponding DSL definition (i.e. abstract syntax, concrete syntax and semantics), and a set of dependencies with other DSLs of the family, which are described at the abstract syntax level, either using import or merge. An important advantage of using MDE techniques is that regular operations on models can be used to manipulate a member. It can be stored in a model repository or serialized, it can be validated using any compatible tool, it can be transformed, and so on.

We argue that to tackle the creation of a family of DSLs, composition mechanisms are needed at the concrete syntax, abstract syntax and semantics levels. Figure 5 shows the relationships between the artifacts involved in the definition of our family of DSLs. When language composition is considered (i.e. reuse at the DSL developer level) the composition is needed for the components of two or more DSL's definitions. It is performed by composing metamodels, concrete syntax definitions and model transformation definitions, in order to get a single, resulting DSL definition. When program composition is considered (i.e. reuse at the DSL user level) an independent DSL definition is attached to each member, but the dependencies between them must be represented. They are established between the abstract syntax metamodels, and must be propagated to the models. The concrete syntax of each member must provide the user with a means to connect the DSL programs. The composition is therefore needed at the DSL program level, that is, the actual composition is performed on models. The concrete syntax to abstract syntax

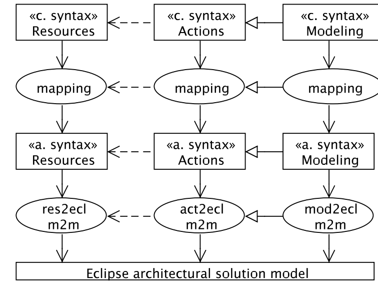

Fig. 5. Composition relationships between the members of the DSL family used as running example, and their implementation artifacts. Dashed arrows means dependency, while empty arrows means extension. *"c.syntax"* means concrete syntax, *"a.syntax"* means abstract syntax, while *"m2m"* means model-to-model transformation.

mapping will be in charge of actually connecting the models. The model transformation definition attached to each member must have the dependencies into account.

In the example, the abstract syntax of the Modeling Actions DSL is created as an extension of the abstract syntax of the Actions DSL. The same applies for the concrete syntax. Regarding model transformations, *mod2ecl* is based on *act2ecl* (see Figure 5). The result of composing all the components is the actual Modeling Actions DSL definition. At the same time, the Resources and Actions DSLs are two different DSLs from the user point of view, but they have dependencies. A dependency is first described in the abstract syntax metamodel, and must be propagated to the rest of the components. For instance, the *act2ecl* transformation definition must be able to get elements created by the *res2ecl* transformation definition.

In the next two subsections we explain how to address importation and extension in a family of DSLs using the composition mechanisms introduced in the previous section.

B. Importation

As with general purpose languages, decomposition of DSL programs into reusable parts, that can be later composed, is a way of improving reuse and avoiding code duplication. Also, separating a DSL program into several files is a way of dealing with the complexity of large specifications. A DSL program is then seen as a reusable module, and mechanisms to compose a module into other modules must exist.

One way to achieve such goals, is to allow a DSL program to import another DSL program, making the elements it defines available. In a family of DSLs this means that dependencies between family members has to be resolved.

At the abstract syntax level we represent dependencies as cross-references between metamodels using the package import relationship presented in Section IV-A. When a metaclass of a family member depends on a metaclass "belonging" to another family member, a cross-reference between metamodels is established. In this way, each DSL program (i.e. a model) conforms to the corresponding family member metamodel, and the dependencies at the abstract syntax level will be propagated as cross-references to other models. Figure 6 shows that in the running example the `ActionGroup` metaclass is related to `ProjectType`, and `Action` is related to `ResourceType`. Any program defining actions has to reference resources defined in another program.

At the concrete syntax level there are two problems involved. First, an "import" statement is needed to allow the user to load

Preliminary Version

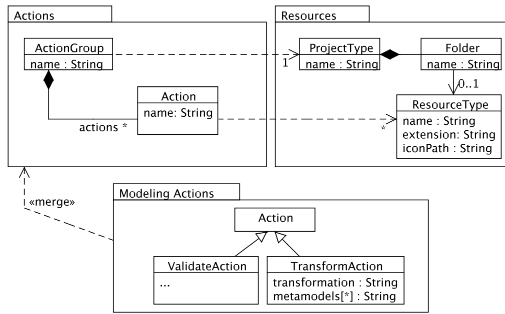

Fig. 6. Excerpt of the metamodels involved in the example family of DSLs, and their relationships. Dashed arrows means a reference to another metamodel, and thus a metamodel importation relationship is established. The merge relationship is indicated with the <<*merge*>> stereotype.

a certain DSL program. This raises the following issue, given a certain DSL program which is going to be imported, how to know which is its definition. We rely on Uniform Resource Identifiers (URI) to uniquely identify DSLs within a family. The name of the DSL program to be imported is prefixed with a logical scheme which identifies its definition (i.e. the member of the family).

In Section IV-C we explained that identifiers are used to reference elements in a non-containment relationship. The importation mechanism makes elements directly available, so that they can be referenced by its identifier as if they had been defined in the same DSL program. However, to avoid name clashes the import statement requires a qualifier to reference the elements defined by the imported program. This means that the imported program becomes the scope of the elements it defines.

The example below shows the DSL program to define actions presented at the beginning of Section III rewritten to consider the importation mechanism. It imports the DSL program to describe resources (let's call it `age-resources`). The `age-resources` program is imported as `res`, so that its elements must be referenced with the `res` prefix. In this way, the `project` keyword references the `age` project type, and `resource` keyword references the resource type identified by `rubytl-transf`, which belong to the DSL program identified as `res`.

```
import 'resources://age-resources', :as => 'res'

action_group 'age-group' do
 project 'res.age'
 action 'export transformation' do
  resource 'res.rubytl-transf'
  show_in_popup
 end
end
```

This is supported by our DSL for mapping concrete to abstract syntax, which resolves qualified cross-references to elements defined in imported DSL programs. To do so, a construct called *import for* is added, so that it is possible to specify that one DSL can import other DSL, and the URI that will be used to identify the imported DSL programs. In this way, one only needs to declare which DSLs can be imported, and to establish the relationships between DSL elements using the *ref* construct.

The following excerpt corresponds to the definition of the Actions DSL. It uses the `import_for` statement to establish an import relationship to the Resources DSL. References to keywords defined in the Resources DSL must also be prefixed with `res`, to avoid name clashes (e.g. a `project` keyword is defined in both Resources and Actions DSLs with different purposes: to define a project, and to reference a project respectively). The implementation resolves the cross-references transparently.

```
import_for 'dsl://resources', :as => 'res' do
  uri 'resources'
end

mappings do
  ...
  ref 'action_group.project' => 'res.project'
end
```

At the model transformation level, the challenge is how to associate an indepedendent transformation definition to each family member, but still allowing dependencies to be resolved. Given a family with DSL_1 and DSL_2 their associated transformations, T_1 and T_2, have to be related when some piece of target model created by T_1 is needed to complete T_2. Our solution relies on using the *trace query* function presented in Section IV-D. In this way, T_2 uses *trace query* to reference elements created by T_1. The abstract syntax model of DSL_1 has elements that are referenced by DSL_2, and *trace query* provides a means to get the corresponding target element, created by T_1, for a given source element "shared" by DSL_1 and DSL_2.

In the running example, the *res2ecl* transformation definition is in charge of dealing with resources. In particular it establishes the mapping between `ResourceType` and `FileResource` (see example of Section IV-D). The *act2ecl* is in charge of dealing with actions (e.g. it may map an `Action` to a `PopupMenuEP`). However, as can be seen in Figure 3, a `PopupMenuEP` is related to a `FileResource`, but to respect the separation of concerns, the *act2ecl* transformation cannot create resource-related elements, but only reference them. In this way, the reference is established in two steps: (1) *res2ecl* is first executed to create resource-related elements, and (2) in *act2ecl* a *trace query* call is performed for each cross-reference between abstract syntax models that must be resolved (i.e. to obtain the corresponding target element).

An excerpt of the *act2ecl* transformation definition is shown below. The only rule shown creates an extension point to define a pop-up menu from each action. The *trace query* call takes the value of the `action.resource` source element, and returns one element of type `FileResource` (i.e. notice the use of `one_of` to ensure that only one element is returned), which was previously created from the source element by the *res2ecl* transformation.

```
transformation 'act2ecl'

rule 'action2popup' do
  from ResDSL::Action
  to   Arch::PopupMenuEP
  filter { |action| action.is_popup? }
  mapping do |action, ep|
    ep.resource = trace_query(action.resource).
                    one_of(Arch::FileResource)
  end
end
```

As part of its phasing mechanism, RubyTL provides a way to define a new transformation definition which is actually the result of composing two or more transformations, which are seen as phases. As explained, existing transformation definitions can be imported within a new one, establishing then the execution order using the `scheduling` construct. In our example, *act2ecl* is executed after *res2ecl* because the former depends on the piece of target model generated by the latter (i.e. the trace query call

Preliminary Version

establishes the dependency).

```
import 'm2m://res2ecl', :as => 'resources'
import 'm2m://act2ecl', :as => 'actions'

scheduling do
  execute 'resources'
  execute 'actions'
end
```

C. Extension

Extending a DSL consists of creating a new DSL that contains the same constructs as the extended DSL, but also some new constructs and new functionality. This allows common parts to be reused in a family of languages: the commonalities of several DSLs can be factorized into a base DSL, that will be later extended with variants.

In the running example, the Actions base language is defined to represent generic actions in the Eclipse platform. It can be reused to describe actions typical of specialized environments built on top of Eclipse. For instance, launching a transformation or validating a model are actions commonly found in a modeling environment. Thus, the Modeling Actions DSL extends Actions to provide such specialized actions.

The following piece of code shows an excerpt of a Modeling Actions DSL program, which defines actions that can be applied to a UML class model in order to transform it to Java code. They are enclosed into a group of related actions. Both `transform_action` and `validate_action` are specialized keywords not available in the Actions DSL. Moreover, `transformation` and `metamodels` are keywords only available for transformation actions.

```
action_group 'uml-transformations' do
  project 'uml-modeling'

  transform_action 'uml-accessors' do
    resource 'uml-files'
    show_in_popup

    transformation 'uml2accessors.rtl'
    metamodels 'UML', 'Java.ecore'
  end

  validate_action 'uml-class' ...
end
```

At the abstract syntax level extension is implemented as a merge relationship between the base metamodel and the corresponding extension metamodel, which adds new concepts and extends some of the existing ones. Creating a generalization relationship from an existing metaclass belonging to the extended metamodel is the way to connect the two DSLs. Additionally, we provide a merge transformation that will create a new composite metamodel resulting from merging both metamodels, as explained in Section IV-A. Figure 6 shows how the Actions DSL abstract syntax is reused by Modeling Actions DSL abstract syntax. In particular, the join point is the `Action` metaclass.

At the concrete syntax level our framework supports extension by means of the concept of keyword extension. In this way, extending a DSL simply consists of importing it into the new DSL, so that its keyword definitions are part of the new DSL, and extending the desired keywords according to the abstract syntax. A generalization relationship is therefore mapped to a keyword extension relationship. The implementation internally merges both concrete syntax definitions, automating the process of creating subclasses of sandboxes explained in Section III-E.

In the code excerpt below, the Actions DSL is extended to create the Modeling Actions DSL. The Actions DSL definition is first imported, and then `transform_action` is defined as an extension of the `action` keyword defined in Actions.

```
extension_for 'dsl://actions', :as => 'act'

keyword 'transform_action', :extends => 'act.action'
keyword 'transformation' do
  param 'filename', :string
end
composition_for 'transform_action' do
  nested 'transformation'
end
```

At the model transformation level the problem involved is how to extend an existing transformation definition, that deals with one family member, in order to add those rules and bindings that are needed to implement the translational semantics of another member, which is an extension of the former.

In the running example, the Modeling Actions DSL is an extension of the Actions DSL, so the *mod2ecl* transformation definition is written as an extension of *act2ecl*. This means that new rules may be added, while others may need to be refined. In particular, the `action2popup` rule shown above must be refined in order to deal with specialized kind of actions, for instance a `TransformationAction` that has a particular Eclipse implementation represented with the `TransfActionImpl` metaclass.

A refinement rule provides us with a means to refine a mapping established by a rule in a previous transformation. Thus, we propose importing the extended transformation into the extending transformation, and using refinement rules to extend the mappings it establishes.

In the transformation excerpt below, the *mod2ecl* transformation definition imports the *act2ecl* transformation definition. Then, a refinement rule "captures" all mappings between `TransformationAction` elements and `PopupMenuEP` elements, allowing them to be refined, in this case to create a new `TransfActionImpl` element and connect it to an existing `PopupMenuEP` element. Implementation-wise the transformation code is enclosed within a *phase* declaration to allow it to be scheduled along with *act2ecl*.

```
transformation 'mod2ecl'
import 'm2m://act2ecl', :as => 'actions'

phase 'modeling-actions' do
  refinement_rule 'action2popup' do
    from ResDSL::TransformationAction
    to   Arch::PopupMenuEP
    mapping do |action, ep|
      ep.impl_class = Arch::TransfActionImpl.new
      ...
    end
  end
end

scheduling do
  execute 'actions'
  execute 'modeling-actions'
end
```

Figure 7 summarizes the techniques we propose to create a family of DSLs. We use *extension* to address language composition. Abstract syntax metamodels are composed using a merge transformation, and the notion of keyword extension is used to

Preliminary version

	Extension	Importation
Abstract syntax	Merge	Import
Concrete syntax	Keyword extension	Qualified identifiers
Trans. semantics	Refinement rule	Trace query

Fig. 7. Summary of techniques to address modularity for the development of a family of DSLs.

represent the generalization relationships at the concrete syntax level. The refinement rule mechanism allows us to extend existing mappings. We tackle program composition with *importation*. The dependencies between family members are represented at the abstract syntax level using an import mechanism to establish cross-references. Qualified identifiers are used to reference elements defined in imported DSL programs. Finally, the trace query operator resolves dependencies by allowing a transformation to get elements created by another transformation.

VI. Discussion

From our experience with embedded DSLs during the development of the AGE tool[3], we present a discussion about the advantages and disadvantages of the embedded approach for DSL development. We conclude with some guidelines to decide when this approach could be a good choice to implement a DSL.

A. Advantages and disadvantages

The main advantage of embedded DSLs is rapid development [11]. A particular strength is that they reuse the infrastructure of the host language, which allows us to have all features of a general purpose language for free. We have taken advantage of this in RubyTL to provide the imperative nature of the language.

The flexibility of the embedded approach (in particular when combined with dynamic languages) allows novel features to be incorporated to a DSL without much implementation effort. This has allowed us to experiment with model transformation language features [28] and DSL composition.

Regarding usability, IDE support is an important issue to make the adoption of a DSL by a community easier. Nowadays, IDEs providing features such as syntax highlighting, code folding, autocompletion, cheat sheets, etc. are common for general purpose programming languages. An embedded DSL not only inherits the host language's features, but one can also take advantage of some features available in existing IDEs for the host language. For instance, features such as syntax highlighting, code folding or even some form of autocompletion are straightforward to reuse.

However, there are some disadvantages in embedded DSLs. The more obvious is that the syntax of the DSL is determined by the syntax of the host language. Since the syntax could not be the optimal one, a domain expert may be unable to use it.

The fact that all features of the host language are available can be a drawback instead of an advantage. In our experience, users are reluctant to use embedded DSLs because they tend to think that it implies learning a new general purpose language. Also, it is not always possible to keep developers working on the wanted abstraction because they may rely on host language's features.

Finally, concerning IDE support, providing autocompletion based on the domain constructs is complicated to implement because it would imply dealing with the whole grammar of the host language.

B. Guidelines

Based on the discussion above we identify several situations where creating an embedded DSL is a suitable option. When the host language is known by the user community, there is no problem to adopt this approach since users are comfortable with the syntax and constructs of the DSL. On the other hand, managing a global community non-knowledgeable in the host language to accept and embrace an embedded DSL is difficult, and a good adoption is improbable. Users tend to think that all features of the host language must be learned, and feel themselves overwhelmed by the "fictitious" need of learning a new language. Moreover, languages well-suited to create embedded DSLs, such as Haskell or Ruby, usually belongs to paradigms the average developer is not used to. Finally, in the case of local user communities a knowledgeable person may train a small team. If the DSL is simple enough, only a minimum initial effort is needed to overcome the initial reluctance.

We have identified four usage scenarios for embedded DSLs. The first, and more obvious, scenario is to develop an embedded DSL intented to be widely used by a certain host language community (e.g. Parsec in Haskell, Rake in Ruby, etc.)

Another scenario is to develop an embedded DSL to automate repetitive tasks usually carried out by a developing team. The running example used in this paper corresponds to such an scenario. Defining a generative family of DSLs, as explained in this paper, is particularly important to improve reuse. If some people of the team do not know the host language, they can be trained by the people in charge of the implementation.

The third scenario we consider is experimentation. The flexibility provided by embedded DSLs is very suitable to experiment with novel features of DSLs. In addition, it allows us to have a working DSL faster than creating the DSL from the scratch using other techniques. In our case, we have experimented with model transformations using this approach. As a result, we have created the RubyTL transformation language, which includes phasing, which is an innovative composition mechanism. In this case, all people involved must know the host language.

Prototyping is an scenario where the embedded DSL approach can be particularly useful. Even when the DSL will be used by a global community, it is still possible to take advantage of embedded DSLs. They can be used to build a prototype of the whole DSL, so that the designer can test those language features which may not be clear. Once the DSL design is complete, the development can switch to a more traditional approach. Another approach is to concentrate in the abstract syntax and semantics of the language, relying in the embedded approach to give the concrete syntax so that the language can be tested in their first stages. In a last stage, an stable concrete syntax can be given using any other approach.

Finally, we remark on the two main reasons difficulting the adoption of an embedded DSL by a user community: tool usability and the lack of experience in the host language. To widen the spectrum of possible users of an embedded DSL, IDE's should provide more usable editors taking into account the use of domain specific constructs, and proper training in the selected host language must be provided, so that developers are able both to use and create embedded DSLs

[3]The AGE tool has been released as free software. It can be downloaded from *http://gts.inf.um.es*.

Preliminary version

VII. Related work

Techniques to embed a DSL into some host language have been discussed in several works. In [7], Hudak coined the term domain-specific *embedded* language to describe such DSLs implemented on top of a host language. This approach has been widely used in Haskell [29] [30] [16]. In Lisp, as argued in [8], defining layers of languages using the macro system is the standard way of dealing with complexity. Building embedded DSLs is also common in dynamic object oriented language communities, such as Ruby and Smalltalk [11] [9] [24]. Embedded DSLs has also been developed in other languages, such as ML [31], Java [19] or C++ [32]. The profiles mechanism of UML can even be considered a means to define an embedded domain modeling language [21] into UML.

With regard to combining the embedded approach with the MDE paradigm, as far as we know, this is the first work addressing this issue. Anyway, in [16] an approach to compile embedded domain specific languages within Haskell is presented. Functional combinators are defined, which create an abstract syntax tree, which is then used to generated code. This is, in some aspects, similar to our proposal, but we rely in a metamodel to define the DSL's abstract syntax, and in a model transformation language for the translational semantics.

In the case of model-based approaches to create textual DSLs, two main categories can be distinguished, grammar-based and metamodel-based. Grammar-based approaches [33] are oriented to generate metamodels from grammars, whereas metamodel-based approaches [2] [17] work on the opposite direction. Developing simple DSLs with these approaches is relatively easy, but the former has the disadvantage of the poor quality of the generated metamodels, while with the latter it can be difficult to fully customize the concrete syntax. In our case we establish a explicit bridge between concrete syntax constructs (i.e. keywords, keyword composition, etc.) and abstract syntax metamodel elements by means of a mapping DSL. This gives freedom in both the concrete syntax and the metamodel structure. In the previous section we have highlighted several advantages of embedded DSLs, but in comparison with other approaches, certainly the most important advantage is flexibility. We have been able to develop a practical approach to create families of DSLs using DSL composition, relying on embedded DSL techniques. Comparing to [17] and [2] this is a distintive aspect. In fact, to our knowledge, our framework is the only one, within those supporting MOF or Ecore metamodels, providing composition mechanisms at the concrete syntax and translational semantics level. On the other hand, as we have pointed out, the embedded approach is not suitable for all scenarios, and then we must rely on some of the former approaches.

The topic of DSL composition has not been widely addressed in the literature. In [13] the problem of building families of DSLs, and how to reuse DSL assets is addressed. It discusses an approach to composing DSLs which is based on abstract syntax templates. However, concrete syntax and transformation composition are not treated. GME is a generic modeling environment which supports composition of metamodels [5] [34]. GME is not based in MOF, but it extends UML with composition operators. The concrete syntax of DSLs defined with GME is graphical, and it is highly coupled to the underlying metamodel definition. In this way, composition at concrete syntax level is based on this extended metamodel composition operators. Metamodel-based assembly techniques has been applied to the field of Situational Method Engineering to create project specific methods from method fragments [35].

Creating a pipeline of DSL translators, each one in charge of translating the corresponding piece of DSL program, is mentioned in [36] as an approach to deal with DSL composition in a family of DSLs. The example given is the *troff* system for text processing. However, this approach works at "the text level", and translators communicate just by manipulating the input text and passing down the result to the following in the chain. Our approach establishes relationships at the abstract syntax level, and it additionally allows reuse to be achieved at the program level (i.e. DSL programs can be composed using importation to related them). In [13] [37], a product line approach is used to create individual languages from language variants. A family of languages is therefore seen as a set of individual languages with some common parts, which are reused. The techniques we propose can be used to create individual languages, but our definition of family of DSLs also covers the creation of related DSLs intended to allow the user to reuse DSL programs (in [12] both approaches are discussed).

Regarding model transformation composition, several approaches at rule-level has been proposed [26] [38]. However, defining families of DSLs requires transformation composition to be addressed with coarser-grained mechanisms, that allow complete transformation definitions to be reused. In this way, we have proposed a phasing mechanism which allows composition of model transformation definitions.

VIII. Conclusions

With the emergence of model driven development, the creation of domain specific languages is becoming a fundamental part of language engineering. The development cost of a DSL should be modest, compared to the cost of developing a general-purpose programming language. Techniques to allow DSLs to be defined without much implementation effort, but focusing on domain aspects are needed. Reuse techniques in the context of DSLs are also a key aspect for DSL approaches to be really effective.

In this paper, we have presented an approach to integrate embedded domain specific languages in a model driven development environment. On the basis of this approach we have tackled the development of families of DSL, as form of reuse, by defining mechanisms for DSL composition. We have shown that creating a family of DSLs requires composition mechanisms at all the levels of a DSL's definition.

Our main contribution is two-fold. On the one hand, we have explained how embedded DSLs can be developed in a systematic way, separating abstract syntax, concrete syntax and semantics using the MDE principles. We have shown that combining embedded DSLs and MDE leverages both of them. On the other hand, we have tackled the development of families of DSLs to deal with DSL reuse. To our knowledge this is the first proposal addressing reuse both from the DSL developer and user point of views, providing a full stack of composition mechanisms.

Regarding the future work, we are investigating other mechanisms for DSL composition, as well as studying how to apply product-line techniques to further improve reuse. We are also researching into evolution of DSLs, by automatically generating transformations intended to adapt both DSL definition artifacts and programs.

Preliminary version

REFERENCES

[1] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, 2005.

[2] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez, "Model-based dsl frameworks," in *Companion to the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, OR, USA*. ACM, 2006, pp. 602–616.

[3] T. Clark, A. Evans, P. Sammut, and J. Willams, *Applied metamodelling: A foundation for language driven development*, September 2004.

[4] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] Ákos Lédeczi, Árpád Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[6] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling: Enabling full code generation*. Wiley-IEEE Computer Society Press, 2008.

[7] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, p. 196, 1996.

[8] P. Graham, *On Lisp: Advanced Techniques for Common Lisp*. Prentice Hall, 1994.

[9] S. Ducasse, A. Lienhard, and L. Renggli, "Seaside: A flexible environment for building dynamic web applications," *IEEE Softw.*, vol. 24, no. 5, pp. 64–71, 2007.

[10] D. Thomas and A. Hunt, *Programming Ruby: The pragmatic programmer's guide*. Addison-Wesley, 2000.

[11] J. S. Cuadrado and J. G. Molina, "Building domain-specific languages for model-driven development," *IEEE Softw.*, vol. 24, no. 5, pp. 48–55, 2007.

[12] A. Evans, G. Maskeri, P. Sammut, and J. S. Willans, "Building families of languages for model-driven system development," in *2nd Workshop on Software Model Engineering (WiSME'03)*, 2002.

[13] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[14] F. Budinsky, S. A. Brodsky, and E. Merks, *Eclipse Modeling Framework*. Pearson Education, 2003.

[15] *Meta Object Facility (MOF) 2.0 Core Specification*, Object Management Group, Inc., oct 2003.

[16] D. Leijen and E. Meijer, "Domain specific embedded compilers," in *DSL'99: Proceedings of the 2nd conference on Conference on Domain-Specific Languages*. Berkeley, CA, USA: USENIX Association, 1999.

[17] A. Kleppe, "Towards the generation of a text-based ide from a language metamodel." in *3rd European Conference on Model Driven Architecture*, ser. Lecture Notes in Computer Science, D. H. Akehurst, R. Vogel, and R. F. Paige, Eds., vol. 4530. Springer, 2007, pp. 114–129.

[18] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2003.

[19] S. Freeman and N. Pryce, "Evolving an embedded domain-specific language in java," in *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 855–865.

[20] J. S. Cuadrado, J. G. Molina, and M. Menarguez, "RubyTL: A Practical, Extensible Transformation Language," in *2nd European Conference on Model Driven Architecture*, vol. 4066. Lecture Notes in Computer Science, June 2006, pp. 158–172.

[21] OMG, "UML specification, v2.1.2," February 2007.

[22] J. Dingel, Z. Diskin, and A. Zito, "Understanding and improving UML package merge," *Software and Systems Modeling*.

[23] J. v. van Wijngaardeen and E. Visser, "Program transformation mechanics: A classification of mechanisms for program transformation with a survey of existing transformation systems," May 2003.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[25] J. S. Cuadrado and J. G. Molina, "Modularization of model transformations through a phasing mechanism," *Software and Systems Modeling*, 2008.

[26] I. Kurtev, K. van den Berg, and F. Jouault, "Rule-based modularization in model transformation languages illustrated with atl," *Sci. Comput. Program.*, vol. 68, no. 3, pp. 111–127, 2007.

[27] R. Lämmel, "Adding Superimposition To a Language Semantics — Extended Abstract," in *FOAL'03 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002; Technical Report CS Dept., Iowa State Univ.*, G. T. Leavens and C. Clifton, Eds., Mar. 2003.

[28] J. S. Cuadrado and J. G. Molina, "A plugin-based language to experiment with model transformations," in *9th International Conference on Model Driven Engineering Languages and Systems*, vol. 4199. Lecture Notes in Computer Science, October 2006, pp. 336–350.

[29] D. Leijen and E. Meijer, "Parsec: Direct style monadic parser combinators for the real world," Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2001-35, 2001.

[30] P. Hudak, "Modular domain specific languages and tools," in *Proceedings: Fifth International Conference on Software Reuse*, P. Devanbu and J. Poulin, Eds. IEEE Computer Society Press, 1998, pp. 134–142.

[31] S. N. Kamin and D. Hyatt, "A special-purpose language for picture-drawing," in *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. Berkeley, CA, USA: USENIX Association, 1997, pp. 23–23.

[32] K. Czarnecki, J. T. O'Donnell, J. Striegnitz, and W. Taha, "Dsl implementation in metaocaml, template haskell, and c++." in *Domain-Specific Program Generation*, ser. Lecture Notes in Computer Science, C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, Eds., vol. 3016. Springer, 2003, pp. 51–72.

[33] M. Wimmer and G. Kramler, "Bridging grammarware and modelware," *Satellite Events at the MoDELS 2005 Conference*, pp. 159–168, 2006.

[34] A. Lédeczi, G. Nordstrom, G. Karsai, P. Valgyesi, and M. Maróti, "On metamodel composition," in *Proceedings of the 2001 IEEE International Conference on Control Applications (CCA)*, 2001, pp. 756–760.

[35] S. Brinkkemper, M. Saeki, and F. Harmsen, "Meta-modelling based assembly techniques for situational method engineering," *Information Systems*, vol. 24, no. 3, pp. 209–228, May 1999. [Online]. Available: http://dx.doi.org/10.1016/S0306-4379(99)00016-2

[36] D. Spinellis, "Notable design patterns for domain specific languages," *Journal of Systems and Software*, vol. 56, no. 1, pp. 91–99, Feb. 2001.

[37] M. Voelter, "A family of languages for architecture description," in *8th OOPSLA Workshop on Domain-Specific Modeling (DSM'08)*, Oct. 2008.

[38] M. Belaunde, "Transformation Composition in QVT," in *Proceedings of the First European Workshop on Composition of Model Transformations*, July 2006, pp. 45–52.

Jesús Sánchez Cuadrado Jesús Sánchez Cuadrado is a PhD candidate at the University of Murcia. His research interests are model-driven development, model transformation languages, and dynamic languages. He received his masters in computer science from the University of Murcia. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jesusc@um.es.

Jesús García Molina Jesús García Molina is a professor of software design at the University of Murcia, where he leads the Software Technology Research Group. His research interests include model-driven development, domain-specific languages, and software processes. He received his PhD in science from the University of Murcia. Contact him at the Dept. of Computers and Systems, Facultad de Informática, Univ. of Murcia, Murcia 30071, Spain; jmolina@um.es.

Preliminary Version