Towards the Systematic Construction of Domain-Specific Transformation Languages

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

Universidad Autónoma de Madrid (Spain)

Abstract. General-purpose transformation languages, like ATL or QVT, are the basis for model manipulation in Model-Driven Engineering (MDE). However, as MDE moves to more complex scenarios, there is the need for specialized transformation languages for activities like model merging, migration or aspect weaving, or for specific domains of wide use like UML. Such *domain-specific transformation languages* (DSTLs) encapsulate transformation knowledge within a language, enabling the reuse of recurrent solutions to transformation problems.
Nowadays, many DSTLs are built in an ad-hoc manner, which requires a high development cost to achieve a full-featured implementation. Alternatively, they are realised by an embedding into general-purpose transformation or programming languages like ATL or Java.
In this paper, we propose a framework for the systematic creation of DSTLs. First, we look into the characteristics of domain-specific transformation tools, deriving a categorization which is the basis of our framework. Then, we propose a domain-specific language to describe DSTLs, from which we derive a ready-to-run workbench which includes the abstract syntax, concrete syntax and translational semantics of the DSTL.

1 Introduction

Model transformations are central to MDE. Many transformation languages have been proposed and are widely used nowadays, e.g. ATL or QVT. We term these languages General-Purpose Transformation Languages (GPTLs), because their scope considers transformation of models, but they are not specific for particular tasks (like migration or refactoring) or domains (like UML or Petri nets).

We can use GPTLs to tackle a wide variety of scenarios, but in our experience, some transformation tasks become more natural and easier by using specialized transformation languages offering primitives tailored to the task to be solved. Examples of these tasks include model migration, promotion of models into meta-models, and aspect weavers for domain-specific languages. Similar to the benefits of using domain-specific languages over general-purpose ones in well-known domains, we claim that these transformation scenarios would benefit from Domain-Specific Transformation Languages (DSTLs). This is so as DSTLs make explicit domain knowledge that otherwise needs to be repeatedly embedded in transformations built with GPTLs or programming languages like Java.

Some works in the literature recognise the need for DSTLs [16, 25, 30]. However, there is a lack of methods and tools for their systematic engineering, including the definition of their abstract syntax, concrete syntax and semantics.

While the design of domain-specific modelling languages is well understood and there is a plethora of workbenches to speed up their construction, this support is lacking for DSTLs. By offering such support, many transformation tasks can be recasted as DSTLs instead of relying on ad-hoc solutions.

In this paper, we propose a design process and tool support for the systematic construction of DSTLs. First, we provide a suitable language to describe the DSTL abstract syntax. This language includes transformation-specific constructs like Mapping, ImperativeRule and Guard. From the description of the DSTL abstract syntax, we generate a MOF-based meta-model which is instantiated to describe concrete transformations, and an initial concrete syntax for the DSTL, tailored to the selected transformation constructs. Depending on the style of the DSTL (e.g. mapping-based or imperative), we also generate a scaffolding of the compilation into the Eclectic transformation virtual machine [7]. Eclectic is a family of transformation languages with different styles (e.g. target-oriented or mapping), and the languages to compile to are selected based on the primitives used in the DSTL description. Instead of relying on code generation to produce Eclectic code, we use model transformations, using a novel template-based technique. We illustrate our proposal with a DSTL for promotion transformations, showing the benefits w.r.t. a hand-made implementation of the DSTL.

Altogether, the contributions of the paper are: (i) the identification of domains and tasks where DSTLs make sense, based on a review of the literature, and (ii) a systematic process for the integral definition of DSTLs.

The paper is organized as follows. Sec. 2 presents an overview of different transformation tasks that would benefit from DSTLs, exposing useful features in each scenario. Sec. 3 introduces our approach and a running example. Sec. 4 presents our way to define the abstract and concrete syntax of DSTLs, while Sec. 5 explains how we specify their semantics. Sec. 6 shows tool support. Sec. 7 reviews related works and, finally, Sec. 8 concludes.

2 Domain specific transformation languages

A DSTL is a transformation language designed for a specific transformation task (e.g. model merging), or restricted to work on special kinds of models (e.g., UML models). Its aim is not to be "universal", applicable to any transformation task, as languages like ATL or QVT are. On the contrary, DSTLs contain domain-specific primitives enabling a more succinct and intensional expression of the task to be performed, which frequently leads to simpler transformation models. Fig. 1 shows a scheme with the main features we require for DSTLs: restricted application context (fixed source or target) or expressivity (e.g. model migration), and syntax tailored to the specific task.

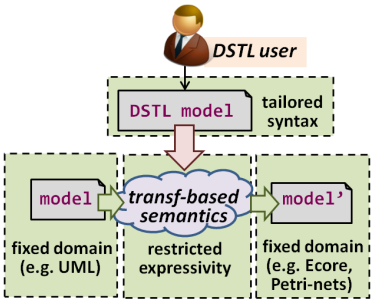

Fig. 1. DSTLs features.

From the literature, we have identified two scenarios for DSTLs: a) transformation tools identified as DSTLs by their creators, built ad-hoc; and b) families of transformation tasks developed using GPTLs, which can be seen as an application area for creating a DSTL. In the first scenario, DSTLs are given semantics by building an interpreter, or by compiling into a general-purpose programming language (GPL), a GPTL or a virtual machine. Moreover, in either scenario, the source or target meta-model of the DSTL may be fixed. For instance, many applications have the recurrent need to transform from a variety of languages into a fixed one, like PROMELA or Petri nets [36] for verification purposes. The second scenario typically arises due to the recurrent need to transform from widely used languages, like UML or XML. These transformations are generally written from scratch using GPTLs or GPLs; however, developers would benefit from explicit linguistic support for the features and structure of the fixed language, which could be provided by DSTLs, leading to simpler transformations.

Next, based on a review of the literature (22 articles), we identify some specific transformation tasks where the availability of DSTLs and systematic means for their construction would be beneficial, pointing out the main features expected from such DSTLs. Table 1 shows a summary, gathering the two typical scenarios for DSTLs (DSTLs built in an ad-hoc way, and families of similar transformation tasks), whether the source or target meta-models of the DSTL should be fixed, if the DSTL is in-place, the main DSTL primitives, the kind of concrete syntax, and the language semantics given by the authors (i.e. the DSTL or tool has a compiler or interpreter, or a GPTL or GPL is used for execution).

1. *Promotion.* This kind of transformations transforms a model into a meta-model. When considering a particular meta-modelling technology, the target meta-model is fixed (e.g. Ecore). Most works in the literature [4, 12, 31] encode these transformations using textual GPTLs – like ATL or Tefkat [19] – or directly in Java. Based on these works, useful primitives in these DSTLs would be mappings, bindings and guards.
2. *Migration.* These transformations deal with the update of artefacts upon meta-model changes [6, 15, 26, 34]. The source and target meta-models, which correspond to the original and evolved meta-models, are not fixed *a priori*. The community has identified the particularities and needs for this scenario, proposing different textual DSTLs for migration. For instance, EMFMigrate [34] is compiled into a transformation virtual machine, Epsilon Flock [26] extends the EOL imperative language to add migration-specific behaviour, and COPE [15] defines primitives for meta-model changes. Most of these languages adopt a mapping-based style for migration rules.
3. *Aspects.* Several languages have been proposed to define aspects for a variety of modelling languages, ranging from general-purpose like UML [17, 35], to domain-specific ones like Petri nets [22, 37] and Building Information Models (BIM) [18]. In most cases, aspects are defined using graphical patterns, thus pointcuts require pattern matching while advices and introductions imply pattern creation. In these works, the support for the application of aspects to models is ad-hoc or by compilation to graph transformation systems.

Table 1. Summary of features for DSTLs in different domains.

	fixed src	fixed tar	in-place	main primitives	concrete syntax	semantics
scenario a: domain specific transformation languages, built ad-hoc						
Aspects [17, 18, 22, 37]	√	√	√	pointcut (match) advice (creation) introduction (creation)	graphical patterns	interpreter
Bridges [5, 9]	√	√		mapping queries, expressions	textual rules	interpreter
BIM [30]	√	√		aggregation	tabular	interpreter
Merging [2, 11]				compare merge, compose	graphical mappings / textual rules	interpreter
Migration [6, 15, 26, 34]	∼[a]			migrate differencing (default) copy guard	textual mappings with OCL expressions	interpreter / compilation to VM
Model inst. [1]				CRUD on refs.	tree-based editor	compilation to VM
scenario b: families of recurring transformation tasks						
Abstraction [28, 29]	√	√	∼[b]	pattern-search merge, split filter	textual patterns / graphical patterns / imperative language	GPTL / GPL
HOT[33]	√	√	∼	queries, creation	textual rules	GPTL
Promotion [4, 12, 31]		√		mapping binding guard	textual mappings with OCL expressions	GPTL / GPL
Refactoring [20, 21]	√	√	√	pattern-search merge, split pull, push	graphical patterns / imperative language	GPTL / GPL

[a] EMFMigrate uses a fixed source

[b] Some are in-place, some are out-place

4. *Abstractions and refactorings.* Abstractions are model operations that produce simpler, higher-level views of a model. For example, [29] presents a catalogue of abstractions for workflow languages, and [28] for modelling languages. Abstractions need to identify relevant patterns (like sequences of nodes or connected components), and then filter, aggregate or merge those patterns. Similar needs are found in approaches for model refactoring [21] and model slicing [3]. While a few works use DSTLs [3], most approaches use GPTLs, like graph transformation [20].
5. *Model merging and composition.* These transformations merge two models through their common elements. There are specific DSTLs targeting this scenario [2, 11], with primitives such as compare, merge or compose.
6. *Higher-order transformations (HOTs).* These are transformations of transformations, and they are developed using textual GPTLs or GPLs. HOT development would benefit from a DSTL specialized for a fixed input/output language (the meta-model of the transformation language, like ATL or QVT), and with higher-order primitives for the manipulation of the specific language (recurrent queries, concise creation of complex patterns, etc.). The proposal in [33], which adds a template language and a library of HOT-specific helpers to ATL, could be reified as a DSTL.
7. *Bridging technical spaces.* These works bridge a technical space, like grammarware [5] or databases [9], with the modelling space. Some authors use DSTLs [5, 9] with specialised query languages for this purpose, but they are

built in an ad-hoc manner. Other approaches use injectors and GPTLs (as in many examples of the ATL Transformation Zoo[1]).

8. *Others.* There are works describing DSTLs for other domains, like [30], which proposes a DSTL to calculate the budget for constructing a building from its model and its components' price. Another example is [1], which describes a DSTL for instantiating model templates using feature models. This DSTL permits specifying instantiation rules, like selecting or deleting references, and its semantics is given by a compilation into the ATC virtual machine.

In conclusion, many domain-specific transformations tasks are currently developed using GPTLs or programming languages, but may benefit from having a more specialized transformation language at hand. In fact, as discussed above, for some domains several DSTLs have already been built to facilitate developing such transformation tasks. This indicates a trend, which can be clearly witnessed in the domain-specific aspect languages area for which there is even an established dedicated workshop [10]. These DSTL approaches typically rely on subsets of the features of a GPTL (e.g. mappings, bindings), equipped with domain-specific constructs (e.g. support for special queries or expressions). The implementations range from totally ad-hoc approaches developed from scratch [5, 9] to more systematic ones, e.g., based on compilation to a VM [34]. Most of the approaches try to reuse well-known syntaxes, such as OCL for queries or graph patterns for rewritings, but such syntaxes are normally encoded from scratch (notable exceptions are [26, 34]). Likewise, the tool support quality varies (not shown in the table), but advanced features such as semantic autocompletion for textual editors and debugging support are missing since their implementation cost is high. Hence, we propose an approach to DSTL development in which the DSTL designer can mix and match features found in GPTLs, as well as adding special features of the domain. Then, our approach speeds up the development of the DSTL tooling using a model-driven approach.

3 Overview and running example

Building a DSTL involves the definition of its abstract syntax, concrete syntax (textual, graphical, tabular, or a mix of them), and execution semantics (typically, developing an interpreter or compiler). Additional elements may be needed, such as a scoping mechanism for variables and a type checker to ensure that the types used in a transformation belong to the meta-models being transformed.

These tasks are normally accomplished using an ad-hoc process without specialized tool support [16], which poses several inefficiencies. First, the abstract syntax has to be devised in terms of MOF constructs instead of using native concepts of the transformation domain, like *rule* or *mapping*. Using these primitives facilitates defining the semantics of the DSTL. Secondly, reusing of typical transformation features, like the use of OCL to define navigation expressions or the integration of a type checker for meta-models has to be done manually.

[1] http://www.eclipse.org/atl/atlTransformations/

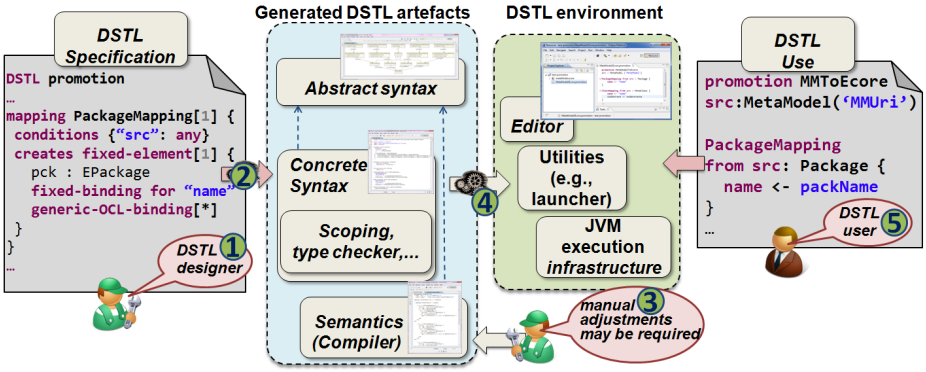

Fig. 2. Our proposal for the construction of domain-specific transformation languages.

Finally, creating an efficient and stable execution infrastructure from scratch is a time consuming and error prone task.

Our solution to these problems is a framework for the development of DSTLs which provides automation for the generation of their abstract syntax, concrete syntax and execution semantics, specialised for the peculiarities of the DSTL. Fig. 2 shows the working scheme of our process. It illustrates the different steps in the construction of a DSTL for *promotion transformations* in order to facilitate the definition of this kind of transformations [4, 12, 31].

As a first step (*label 1*), the DSTL designer defines the primitives to be included in the DSTL. For this purpose, we make available a domain-specific language (DSL) which includes constructs close to the transformation domain. For instance, in the figure, the DSTL specification includes the definition of a mapping type that will be used to identify which source elements will be promoted into packages, and thus, the source of the mapping can be any element but the target is fixed (EPackage). The complete DSTL specification includes additional mapping types for the promotion of the other modelling elements.

Our DSL uses primitives of the transformation domain, facilitating the generation of several artefacts. In particular, from the DSTL specification, we generate an Ecore meta-model for the abstract syntax of the DSTL, a default implementation of its concrete syntax (including a scope resolutor and a type checker for the input/output meta-models), and an initial scaffolding of the compiler (*label 2*). The generated concrete syntax is tailored to the transformation primitives selected in the DSTL. The compiler synthesizes Eclectic code from the DSTL model, and this code is compiled into the Java Virtual Machine.

The designer can customize the default abstract syntax, concrete syntax and compiler (*label 3*), e.g., to add domain-specific functionality. Then, an environment for the language (*label 4*) is automatically created. The figure shows a DSTL user building a promotion transformation using the generated DSTL (*label 5*).

In the following two sections we detail the different steps in our framework.

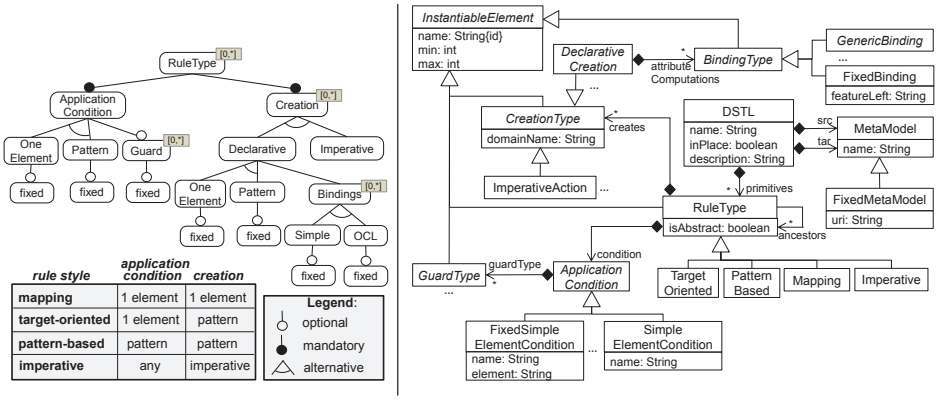

Fig. 3. Rule types (left). Meta-model for specifying DSTLs (excerpt) (right).

4 Designing the syntax

This section presents our DSL to define DSTLs (sec. 4.1), and the subsequent generation of the abstract and concrete syntax (secs. 4.2 and 4.3).

4.1 Describing the DSTL

As we have seen, DSTLs profit from native concepts of the transformation domain, and may combine features and operations found in GPTLs with domain-specific ones. Thus, the first step is choosing the required constructs for the language. For space reasons, in this paper, we focus on the selection and customization of suitable rule types, since it is normally the main construct of a DSTL, although in practice we have considered other aspects such as scheduling and tracing.

The left of Fig. 3 shows a feature diagram describing the kind of rules that could be constructed with our approach. All rule types have an application condition and creation directives. Application conditions identify the elements matched by the rule, which can be either a single element or a pattern, and can include guards. If the DSTL has a fixed source or target meta-model (as is the case in our running example for promotion transformations), then it is possible to use concrete types of the fixed meta-model in the DSTL definition; otherwise, the types are not fixed and should be specified in the transformation. Rule types also need to define the style of the creation directives, which can be either declarative or imperative. In the former case, the creation can be of a single element or of a graph pattern, and it is possible to define bindings for the attributes. For the elements marked with cardinality [0, *] in the feature diagram, the designer can fine-tune the number of times they can be instantiated (any number by default). The most common rule styles result from the combination of the features in this diagram (see the table at the bottom of the figure): *mappings* (rules that create one target element from one source element), *target-oriented rules* (creation of a target graph pattern from a source element), *pattern-based rules* (patterns in the rule source and target), and *imperative rules*.

We have realized these design choices in the meta-model shown to the right of Fig. 3. Its root class is DSTL, which defines if the transformation is in-place and the source and target meta-models. To indicate that a meta-model is fixed in the DSTL, we use the metaclass FixedMetaModel. A DSTL contains transformation primitives, which are instances of RuleType or its subclasses. A RuleType has an application condition (subclasses of ApplicationCondition) and creates elements (subclasses of CreationType). The application condition describes the conditions needed to apply the rule. For instance, FixedSimpleElementCondition is used for rule types having a single source element with a type from the fixed source meta-model. In this case, the rule will be applied to each element of type element. Conditions can also be of type SimpleElementCondition, and in this case, the rule will be applied to each element of the type specified in the specific transformation. Application conditions can have in addition a number of GuardTypes.

The elements a RuleType creates are configured through subclasses of CreationType, like ImperativeAction and DeclarativeCreation. The latter has subclasses for the creation of one or several elements of a fixed or variable type, and can be assigned BindingTypes for attribute computation. Two supported binding types are GenericBinding, where the target feature is specified in the particular transformation, and FixedBinding, which forces the binding of a particular feature of the created element.

The subtypes of RuleType force the use of specific combinations of application conditions, guards, creations and bindings, as shown in the feature diagram. For example, a Mapping has a FixedSimpleElementCondition or SimpleElementCondition, simple declarative creation, and any number of guards and bindings. It is possible to instantiate RuleType if no more specific rule type suits the needs of the DSTL.

Fig. 4 shows part of the definition of the promotion DSTL example, using a concrete syntax we have devised for the previous meta-model. Lines 1–3 declare the DSTL with a fixed target meta-model (ecore). Lines 5–7 declare an abstract mapping with a simple application condition (an element "src" of any available type from the input meta-model). Then, two concrete mapping types are declared that inherit from Common, and hence receive the same condition type. The PackageMapping mapping creates and element with fixed type EPackage from ecore, and has a fixed binding for the feature name and any number of arbitrary OCL bindings. Thus, a transformation using the DSTL will require providing the source type that will be transformed into an EPackage, a literal or an expression for the fixed binding, and any number of additional bindings.

4.2 Generating the abstract syntax

From the description of the desired primitives of the DSTL, we automatically generate a meta-model. As an example, Fig. 5 shows an excerpt of the generated meta-model for our promotion DSTL (package promotion). Each metaclass in this meta-model inherits from some base class in package dstlInst. This package contains infrastructure classes, like Mapping or Binding. For example, a root metaclass Promotion is added to the DSTL meta-model, which inherits from the infrastructure class Transformation. As the source meta-model is not fixed, the

```
1  DSTL promotion
2  src: variable[name: "src" ]
3  tar: fixed[name: "ecore", uri: "http://www.ecl..."]
4
5  abstract mapping Common {
6    conditions { "src": any }
7  }
8  mapping PackageMapping[1] extends Common {
9    creates:
10     fixed−element EPackage[1] {
11       fixed−binding for "name"
12       generic−OCL−binding[∗]
13     }
14 }
15 mapping ClassMapping[1..∗] extends Common {
16   creates:
17     fixed−element EClass[1] {
18       fixed−binding for "name"
19       generic−OCL−binding[∗]
20     }
21 }
22 ...
```

Fig. 4. Describing the promotion DSTL (excerpt).

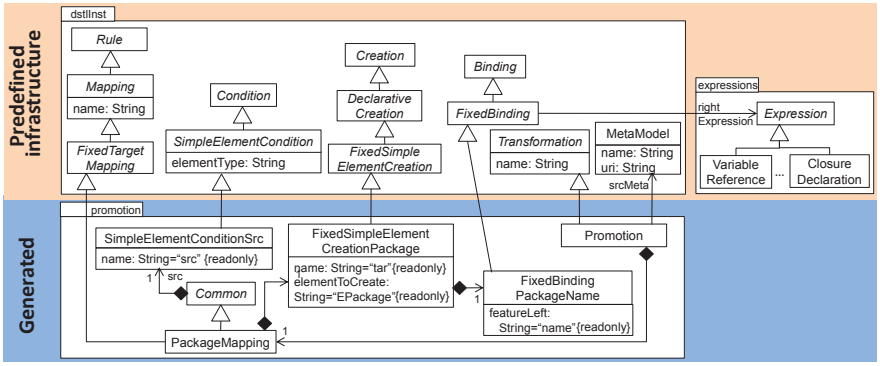

Fig. 5. Generated abstract syntax meta-model (excerpt).

root metaclass includes a reference to MetaModel, to allow the specification of a source meta-model in the specific transformation. From the mapping PackageMapping in the DSTL definition, a metaclass with the same name is added to the meta-model. This metaclass inherits from FixedTargetMapping, which reflects the fact that it is a mapping and its target is fixed. The metaclass also inherits from Common in the same package, which defines the source application conditions for the mapping. Please note that our generator is able to select the base classes to inherit from based on the chosen primitives (e.g. Mapping in Fig. 3), but also on the selected rule features if the more general RuleType metaclass is used instead. For instance, a RuleType with one-element condition and one-element creation is automatically classified as a Mapping. Finally, we have a dedicated infrastructure package for OCL-like expressions. This is used in our running example, since the description of the promotion DSTL includes OCL-like bindings.

The rationale of this approach is that the simple description in Fig. 4 carries semantic information (e.g., on desired rule types), which can be automatically carried to a meta-model (see Fig. 5), and that is used to generate a default textual syntax (see Section 4.3), and a compiler for the DSTL (see Section 5).

4.3 Generating the concrete syntax

From the DSTL description, we also generate a concrete syntax. We support two styles: textual, and tabular for the case of DSTLs consisting of mappings

only. The concrete syntax is based on Xtext, and is customized according to the transformation primitives used (mapping, pattern-based, imperative, etc.). For pattern-based rules, we generate a syntax for patterns similar to that of QVT-Relations. For mappings, we support two styles: one with simple rules (like the syntax of ATL) and another one with nested rules, available if the target meta-model is fixed. Nested rules follow the structure of composite associations of the target meta-model. Finally, if the DSTL uses an expression language (as in Fig. 5), the syntax of the expression language is integrated in the DSTL syntax.

Listing 1 shows an excerpt of a promotion transformation in concrete syntax, using the simple rules style generated by default. This syntax is more concise than e.g., ATL, as we have a fixed target language. Lines 4 and 9 instantiate the package and class mappings, where only the source of the mappings needs to be given. Lines 6 and 11 correspond to the mandatory name bindings.

```
1  promotion MM2Ecore
2  src: MetaModel("MetaModel")
3
4  PackageMapping
5  from src: MetaModel {
6    name <- packName
7  }
8
9  ClassMapping
10 from src: MetaClass {
11   name <- className
12 }
13 ...
```

Listing 1. Excerpt of promotion transformation in DSTL syntax.

```
1  eclectic promotion2ecore (src) -> (ecore)
2  navigation promotion_navigation (src)
3    def src!MetaModel.reachClasses
4      src!MetaClass.all_instances
5    end
6  end
7
8  mappings promotion ( src ) -> ( ecore )
9    from src : src!MetaModel to tgt : ecore!EPackage
10     tgt.eClassifiers <- src.reachClasses
11     tgt.name = src.packName
12   end
13
14   from src : src!MetaClass to tgt : ecore!EClass
15     tgt.name = src.className
16   end
17 end
```

Listing 2. Equivalent Eclectic transformation.

5 Designing the semantics

To define the executable semantics of the DSTL, we establish a mapping to one or more languages of the Eclectic model transformation family [7]. In Eclectic, each language addresses a specific transformation concern, and can be combined with the other languages through composition mechanisms. Eclectic currently provides the following languages: *i)* a mapping language to define one-to-one and one-to-many correspondences, *ii)* a target-oriented language with object notation and explicit rule calls, *iii)* an attribution language to compute inherited and synthesized attributes, *iv)* a pattern matching language with object-notation, and *v)* a lower-level imperative language, which also plays the role of scheduling language and supports in-place transformation. Languages *i-iv* do not allow complex expressions, but these need to be encoded in navigation libraries. The combination of these languages covers many of the scenarios studied in Section 2.

Listing 2 shows the Eclectic transformation with equivalent semantics to the promotion transformation in Listing 1. The *mappings* language is naturally used to establish the correspondences declared in the promotion, where the target type is implicitly given by the rule type. Thus, the Eclectic mapping in lines

9–12 corresponds to the PackageMapping rule, while the mapping in lines 14–16 corresponds to the ClassMapping rule. Bindings for the name property have a direct correspondence in Eclectic. However, the promotion in Listing 1 does not specify how to relate classes to packages, that is, a binding to fill the reference eClassifiers in EPackage is missing. This is "domain knowledge" which gets inferred if no such binding is given. In this case, the default behaviour is generating the binding in line 10 (tgt.eClassifiers ← src.reachClasses), as well as the helper reachClasses within a navigation module which extends the source meta-model (lines 2–6).

To automate the generation of Eclectic code, we need a compiler from the abstract syntax of the promotion DSTL to Eclectic. The next subsection presents a novel facility to support the development of such compilers.

5.1 Code generation by template-based transformations

In the simplest case, a DSTL has a direct mapping to a language in Eclectic. A compiler would be conceptually straightforward to implement, but in practice, it is complex because it requires knowledge of the abstract syntax of Eclectic. An alternative is to generate plain-text (as in [16]), but this neither guarantees the syntactic correctness of the language, leading to a brittle solution, with cumbersome develop-generate-compile-test cycles, nor allows traceability between the DSTL and Eclectic (which is needed for a debugger).

To address this issue, we propose a *template language* targeting Eclectic. This is a model transformation language embedded into the Eclectic syntax, which facilitates specifying model-to-model transformations into Eclectic. In this way, transformation constructs are embedded in Eclectic syntax, like flow control constructs (iteration and condition) and placeholders, in the style of the Model-to-Text standard [23]. Most importantly, it is a safe template system, as the generated Eclectic programs are guaranteed to be syntactically correct. Hence, we obtain the benefits of model-to-text transformations (we use the concrete syntax of the target), and those of model-to-model transformations (we generate syntactically correct models).

Fig. 6 shows an excerpt of the compiler specification for the running example. Lines 2-3 declare the input and output meta-models that the generated transformation will use. The source meta-model in line 2 is obtained from the DSTL model using the expression between "[" and "/]". Line 5 defines a template to map each PromotionProgram to one or more Eclectic modules, so that we can use Eclectic syntax within the template to define a *mappings* transformation and a *navigation* module. The compiler specification does not process a promotion program, but it specifies the rules that the generated compiler will follow.

To create rules for packages, the specification iterates over the refPackageMapping reference to obtain all instances of PackageMapping (line 8). The target type EPackage is fixed (line 10), while the source type is given in the particular promotion transformation, and hence an expression is used (line 9). Lines 11–12 navigate the binding for name and generate the corresponding Eclectic binding. Lines 15-20 check whether a binding for reference eClassifiers has been provided (as it is optional), and if not, they generate a default behaviour to add classes

```
1  compiler spec promotion2ecore
2  input src : [ t.srcMeta.uri /]
3  output ecore : 'http://www.eclipse.org/emf/...'
4
5  [template createTransformation
6   (t : PromotionProgram) : EclecticModule]
7    mappings promotion(src) -> (ecore)
8      [ for m : t.refPackageMapping ]
9        from src : src![ m.src.elementType /]
10       to tgt : ecore!EPackage
11       [ b : m.CreatePackage.nameBinding /] ->
12         tgt.name = src.[ b.rightFeature /]
13
14       // Optional class binding
15       [if b : m.CreatePackage.classBinding ]
16         tgt.eClassifiers = src.[ b.rightFeature /]
17       [else]
18         // Fixed behaviour to resolve classes
19         tgt.eClassifiers <- src.reachClasses
20       [/if]
21       // Generate OCL bindings...
22     [/for]
23     // Similar rules for classes, attributes, etc...
24   end
25
26   navigation promotion_navigation(src)
27    [ for m : t.refPackageMapping ]
28      def src![ m.src.elementType /].reachClasses
29        src![ t.refClassMapping.src.elementType /].
30              all_instances
31      end
32      // Helpers for expressions, OCL bindings...
33    [/for]
34   end
35 [/template]
```

Fig. 6. Excerpt of the compiler specification for the promotion DSTL.

(transformed by ClassMapping rules of the DSTL) to the package. This requires generating a helper method, shown in lines 28-31, because Eclectic only allows complex expressions in navigation modules. This shows that the DSTL designer can provide sensitive defaults to simplify the common cases.

In our proposal, it is important to select the best suited Eclectic languages for the concerns of the DSTL, as a wise choice will facilitate the compiler specification, like in the example of Fig. 6. To help in this process, we have analysed how to map different DSTL features to Eclectic languages. Thus, given a DSTL specification, we generate a scaffolding of the compiler with a selection of languages. For instance, we generate most of the code in Fig. 6, except lines 19 and 28–31, which depend on design decisions of the DSTL designer. Even though we could generate the complete specification, we believe that the power of a DSTL comes from providing domain-specific constructs and sensible default behaviour. Such degree of generation coverage is possible because our DSL to specify DSTLs provides richer semantic information than a plain MOF meta-model.

6 Tool support

We have developed a prototype tool demonstrating our approach. The left of Fig. 7 shows our workbench to build DSTLs, being used to define a more complete version of the promotion DSTL which includes application conditions with arbitrary patterns and guards. From this description, the workbench generates a configuration model (with similar purpose as the *genModel* in EMF) to fine-tune the generation process, e.g., giving names for packages, file extensions, and the type of concrete syntax. Currently, we support three styles for the concrete syntax: textual *simple rules* (like in ATL), textual *nested rules*, and tabular for mapping-based DSTLs. The configuration model contains sensible defaults, so that oftentimes there is no need for any adjustment.

Starting from the DSTL definition, an environment for it is generated using the configuration model. Fig. 7 shows the generated artefacts: an ecore metamodel, a compiler specification, and a fully functional Xtext project. The upper

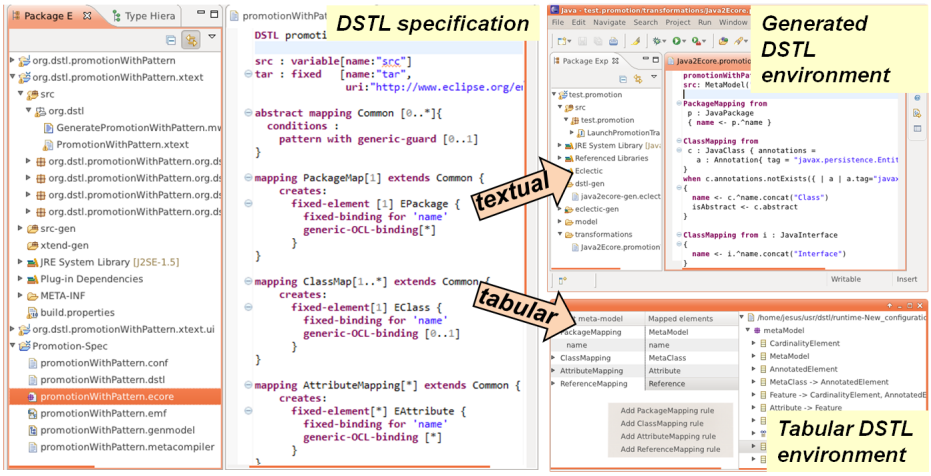

Fig. 7. DSTL workbench and generated environments.

```
1  mappings promotion (src) -> (ecore)
2    uses promotion_pm
3
4    from src : promotion_pm!classMappingPM
5      to tgt : ecore!EClass
6      tgt.name = src.nameExpressionDelegate
7    end
8    ...
9  end
```

```
10 patterns promotion_pm(src)
11   def classMappingPM() -> (c)
12     c : in!JavaClass {
13       annotations = a : Annotation {
14         tag = "javax.persistence.Entity"
15       }
16     }
17   end
18 end
```

Fig. 8. Excerpt of the Eclectic transformation for the example in Fig. 7

right of the figure shows the DSTL environment in action. It offers a customized editor for promotion models, together with a compiler to synthesize Eclectic transformations from DSTL models.

In this example, the chosen rule types have patterns, guards and create fixed target elements. For this reason, we compile into the pattern-matching and mapping languages of Eclectic, which implies creating two transformation modules. To give an intuition of the composition style of Eclectic (detailed in [7]), Fig. 8 shows how the mapping transformation is seamlessly enriched with pattern matching. Line 2 imports the pattern matching module promotion_pm, used as a regular model where patterns are interpreted as types. In this way, the from part of the rule matches the pattern called classMappingPM (line 4), defined in the promotion_pm module (lines 10–18). Patterns use object-diagram syntax.

The tabular syntax shown in the bottom right of Fig. 7 permits configuring mappings by dragging types from the meta-model to the right to the appropriate mapping, and filling the required bindings.

While we are still working on a comprehensive evaluation of our approach, the preliminary results indicate that it can be used effectively to construct DSTL workbenches. As an estimation, from the specification for the running example, declaring 5 types of rules in 43 LOCs (without blank lines), we generate: a meta-model (15 classes and 31 features), a customized Xtext grammar (166 LOCS, 29

rules, and including an expression language), and a compiler specification (49 LOCS), which in turn becomes a compiler (250 LOCs). Thus, the advantage is that from a single compact specification, many heterogeneous complex artefacts are generated and integrated into a ready-to-run customized environment for the DSTL. Moreover, the generated compiler is guaranteed to generate syntactically correct transformations, as it is specified with a safe template system.

7 Related work

Next, we review works targeting the construction of DSTLs. [16] proposes building a meta-model for the DSTL and its compilation into a GPTL (the Epsilon languages) using a model-to-text transformation. However, this work does not provide a systematic approach, or supports defining the abstract and concrete syntax of the DSTL, and there is no traceability between the DSTL models and the generated code. In [25], the authors propose a framework to generate Java-based execution engines for DSTLs, starting from en EBNF grammar. However, there is no description on how this can be achieved in practice.

The Epsilon languages can be seen as a set of DSTLs built atop the Epsilon Object Language [24]. While they leverage from EOL's concrete syntax and semantics, defining a new language needs from a manual extension of the ANTLR grammar and a manual Java encoding of the semantics. Instead, we provide model-driven support for the definition of the abstract syntax, concrete syntax and semantics, which does not restrict the DSTL to any specific concrete syntax, and specialised semantics can be given via compilation.

T-Core [32] is a set of scheduling primitives for model transformation, based on pattern-matching and rewriting. While T-Core's goal is to define flexible rule control languages, our approach describes DSTLs in an integral way. Nonetheless, T-Core's primitives could complement the ones in Eclectic. [27] proposes building DSTLs by mixing the concrete syntax of the involved DSLs (for the rule patterns), together with a transformation language. While that proposal leaves its realization to future work, it could be complemented with our approach, which focusses more on designing the transformation language itself.

Regarding the generation of transformation code, the ACG language is designed to generate bytecode for the ATL VM from a model. However, it is too low-level to use it for implementing DSTLs. Thus, in [33], a template language for ATL is proposed, with no implementation available. Even though still a prototype, our template language for Eclectic is, to our knowledge, the first safe template language to generate transformations (inspired by [13, 14]).

Finally, [8] provided a feature-based survey of different transformation styles. Our work is based on a subset of common features found in GPTLs, focusing on those that are needed to implement DSTLs. Besides, we allow features to be chosen and combined into a DSTL.

8 Conclusions and future work

In this paper, we have proposed a systematic process and support for the creation of DSTLs. From a DSL-based description of the DSTL, several artefacts are generated: an abstract syntax, a (Xtext-based or tabular) concrete syntax and a compiler specification for an Eclectic language, or a combination of them. The proposal is supported by a prototype implementation.

This work opens a wide line of research, similar to the one initiated years ago by works dealing with the automated creation of environments for domain-specific modelling languages. In this respect, we plan to continue improving our prototype, and explicitly consider important aspects of transformations like bidirectionality or scheduling. We will also support other types of concrete syntax, including graphical ones, and plan to extend our template approach.

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139).

References

1. O. Avila-García, A. Estévez, and E. Rebull. Using software product lines to manage model families in model-driven engineering. In *SAC*, pages 1006–1011. ACM, 2007.
2. J. Bézivin, S. Bouzitouna, M. D. D. Fabro, M.-P. Gervais, F. Jouault, D. S. Kolovos, I. Kurtev, and R. F. Paige. A canonical scheme for model composition. In *ECMDA-FA*, volume 4066 of *LNCS*, pages 346–360. Springer, 2006.
3. A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling model slicers. In *MoDELS*, volume 6981 of *LNCS*, pages 62–76. Springer, 2011.
4. M. Brambilla, P. Fraternali, and M. Tisi. A metamodel transformation framework for the migration of WebML models to MDA. In *MDWE'08*, pages 91–105, 2008.
5. J. L. Cánovas Izquierdo and J. García Molina. Extracting models from source code in software modernization. *SoSyM*, pages 1–22, 2012.
6. A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing dependent changes in coupled evolution. In *ICMT*, volume 5563 of *LNCS*, pages 35–51. Springer, 2009.
7. J. S. Cuadrado. Towards a family of model transformation languages. In *ICMT'12*, volume 7307 of *LNCS*, pages 176–191. Springer, 2012.
8. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–646, 2006.
9. O. Díaz, G. Puente, J. L. C. Izquierdo, and J. G. Molina. Harvesting models from web 2.0 databases. *SoSyM*, 12(1):15–34, 2013.
10. Domain-specific aspect languages workshop. `http://www.dsal.cl/`.
11. K. Engel, R. Paige, and D. Kolovos. Using a model merging language for reconciling model versions. In *ECMFA*, volume 4066 of *LNCS*, pages 143–157. Springer, 2006.
12. J. Gallardo, C. Bravo, and M. A. Redondo. A model-driven development method for collaborative modeling tools. *J. Net. Comp. App.*, 35(3):1086–1105, 2012.
13. F. Heidenreich, J. Johannes, M. Seifert, C. Wende, and M. Böhme. Generating safe template languages. In *SIGPLAN Not.*, volume 45, pages 99–108. ACM, 2009.
14. Z. Hemel and E. Visser. PIL: A platform independent language for retargetable DSLs. In *SLE'09*, volume 5969 of *LNCS*. Springer, 2009.
15. M. Herrmannsdoerfer. COPE - a workbench for the coupled evolution of metamodels and models. In *SLE*, volume 6563 of *LNCS*, pages 286–295. Springer, 2010.

16. J. Irazábal, G. Pérez, C. Pons, and R. S. Giandini. An implementation approach to achieve metamodel independence in domain specific model manipulation languages. In *ICSOFT*, pages 62–69. SciTePress, 2012.
17. J. Kienzle, W. A. Abed, F. Fleurey, J.-M. Jézéquel, and J. Klein. Aspect-oriented design with reusable aspect models. *Trans. on AOSD VII*, pages 272–320, 2010.
18. M. Kramer, J. Klein, and J. Steel. Building specifications as a domain-specific aspect language. In *DSAL*. ACM, 2012.
19. M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.
20. T. Mens. On the use of graph transformations for model refactoring. In *GTTSE*, volume 4143 of *LNCS*, pages 219–257. Springer, 2005.
21. T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.
22. T. Molderez, B. Meyers, D. Janssens, and H. Vangheluwe. Towards an aspect-oriented language module: Aspects for Petri nets. In *DSAL*. ACM, 2012.
23. OMG. MOFM2T 1.0. `http://www.omg.org/spec/MOFM2T/1.0/`.
24. R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*, pages 162–171, 2009.
25. T. Reiter, E. Kapsammer, W. Retschitzegger, W. Schwinger, and M. Stumptner. A generator framework for domain-specific model transformation languages. In *ICEIS*, pages 27–35, 2006.
26. L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model migration with Epsilon Flock. In *ICMT*, volume 6142 of *LNCS*, pages 184–198. Springer, 2010.
27. B. Rumpe and I. Weisemöller. A domain specific transformation language. In *Models and Evolution*, 2011.
28. B. Selic. A short catalogue of abstraction patterns for model-based software engineering. *Int. J. Software and Informatics*, 5(1-2):313–334, 2011.
29. S. Smirnov, H. A. Reijers, M. Weske, and T. Nugteren. Business process model abstraction: a definition, catalog, and survey. *Dist. Par. Datab.*, 30(1):63–99, 2012.
30. J. Steel and R. Drogemuller. Domain-specific model transformation in building quantity take-off. In *MoDELS*, volume 6981 of *LNCS*, pages 198–212, 2011.
31. J. Steel, K. Duddy, and R. Drogemuller. A transformation workbench for building information models. In *ICMT*, volume 6707 of *LNCS*, pages 93–107, 2011.
32. E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *ECEASST*, 29, 2010.
33. M. Tisi, J. Cabot, and F. Jouault. Improving higher-order transformations support in ATL. In *ICMT'10*, volume 6142 of *LNCS*, pages 215–229. Springer, 2010.
34. D. Wagelaar, L. Iovino, D. D. Ruscio, and A. Pierantonio. Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In *ICMT*, volume 7307 of *LNCS*, pages 192–207. Springer, 2012.
35. M. Wimmer, A. Schauerhuber, G. Kappel, W. Retschitzegger, W. Schwinger, and E. Kapsammer. A survey on UML-based aspect-oriented design modeling. *ACM Comput. Surv.*, 43(4):28, 2011.
36. U. Winkler, M. Fritzsche, W. Gilani, and A. Marshall. Bob the builder: A fast and friendly model-to-petrinet transformer. In *ECMFA*, volume 7349 of *LNCS*, pages 416–427. Springer, 2012.
37. D. Xu and K. E. Nygard. Threat-driven modeling and verification of secure software using aspect-oriented petri nets. *IEEE TSE*, 32(4):265–278, 2006.