Enabling the Collaborative Definition of DSMLs[⋆]

Javier Luis Cánovas Izquierdo and Jordi Cabot

AtlanMod, École des Mines de Nantes – INRIA – LINA, Nantes, France
{javier.canovas,jordi.cabot}@inria.fr

Abstract. Software development processes are collaborative in nature. Neglecting the key role of end-users leads to software that does not satisfy their needs. This collaboration becomes specially important when creating Domain-Specific Modeling Languages (DSMLs), which are (modeling) languages specifically designed to carry out the tasks of a particular domain. While end-users are actually the experts of the domain for which a DSML is developed, their participation in the DSML specification process is still rather limited nowadays. In this paper we propose a more community-aware language development process by enabling the active participation of all community members (both developers and end-users of the DSML) from the very beginning. Our proposal is based on a DSML itself, called Collaboro, which allows representing change proposals on the DSML design and discussing (and tracing back) possible solutions, comments and decisions arisen during the collaboration.

1 Introduction

The active participation of end-users in the early phases of the software development life-cycle is key when developing software [1]. Among other benefits, the collaboration promotes a continual validation of the software to be build [2], thus guaranteeing that the final software will satisfy the users' needs.

When the software targets a very specific and complex domain, this collaboration makes even more sense. Only the end-users have the domain knowledge required to drive the development. This is exactly the scenario we face when specifying a Domain-Specific Modeling Language (DSML). A DSML is a language specifically designed to perform a task in a certain domain. DSMLs appeared as an alternative to General-Purpose (modeling) Languages (GPLs), like UML, to facilitate the modeling of systems in domains that could not be easily represented using the concepts provided by GPLs.

Clearly, to be useful, the concepts and notation of a DSML should be as close as possible to the domain concepts and representation used by the end-users in their daily practice. Therefore, the role of domain experts during the DSML specification is vital, as noted by several authors [3,4]. Unfortunately, nowadays participaton of end-users is still mostly restricted to the initial set of interviews to help designers analyze the domain and/or to test the language at the end, which requires the development of fully functional language prototypes (including a model editor, a parser, etc.) [4,5]. This long iteration cycle is a time-consuming and repetitive task that hinders the process performance [3]

⋆ This work has been supported by the European Commission under the ICT Policy Support Programme, grant no. 317859.

since end-users must wait until the end to see if designers correctly understood all the intricacies of the domain. A second major problem is the lack of traceability of the design decisions. The rationale behind decisions made during the language specification are implicit so it is not possible to understand or justify why, for instance, a certain element of the language was created in that specific way or given that particular type. This hampers the future evolution of the language.

Existing project management tools such as Trac[1] or Jira[2] provide the environments required to develop collaboratively software systems. These tools enable the end-user participation during the process, thus allowing developers to receive feedback at any time [6]. However, their support is usually defined at file level, meaning that discussions and change tracking are expressed in terms of lines of textual files. This is a limitation when developing DSMLs, where a special support to discuss at language element level (i.e., domain concepts and notation symbols) is required to address the two challenges previously described and therefore promote the participation of the end-users.

In order to alleviate these shortcommings, in this paper we present *Collaboro*, a DSML which enables the involvement of the community (i.e., end-users and developers) in the DSML creation process. The language allows modeling the collaborations between community members taking place during the definition of a new DSML. Collaboro supports both the collaborative definition of the abstract (i.e., metamodel) and concrete (i.e., notation) syntaxes for DSMLs by providing specific constructs to enable the discussion. Thus, each community member will have the chance to request changes, propose solutions and give an opinion (and vote) about those from others. We believe this discussion will enrich the language definition significantly and ensure that the end result satisfies as much as possible the expectations of the end-users. Moreover, the explicit recording of these interactions will provide plenty of valuable information to explain the language evolution and justify all design decisions behind it, as also proposed in requirement engineering [7]. Together with the Collaboro DSML we provide the tooling infrastructure and process guidance required to apply Collaboro in practice.

Paper Structure. Section 2 presents the collaborative process we are proposing while Section 3 the language infrastructure needed. Next, the implemented tool and a case study are described in sections 4 and 5, respectively. Finally, Section 6 reviews the related work and Section 7 draws some conclusions and future work.

2 Making DSMLs Development Collaborative

A DSML is defined through three main components [8]: abstract syntax, concrete syntax, and semantics. The abstract syntax defines both the language concepts and their relationships, and also includes well-formedness rules constraining the models that can be created with the language. Metamodelling techniques are normally used to define the abstract syntax. The concrete syntax defines a notation (textual, graphical or hybrid) for the concepts in the abstract syntax, and a translational approach is normally used to provide semantics, though most of the time it is no explicitly formalized[3].

[1] http://trac.edgewall.org/

[2] http://www.atlassian.com/es/software/jira/overview

[3] The collaborative definition of the DSML semantics is out of the scope of this paper.

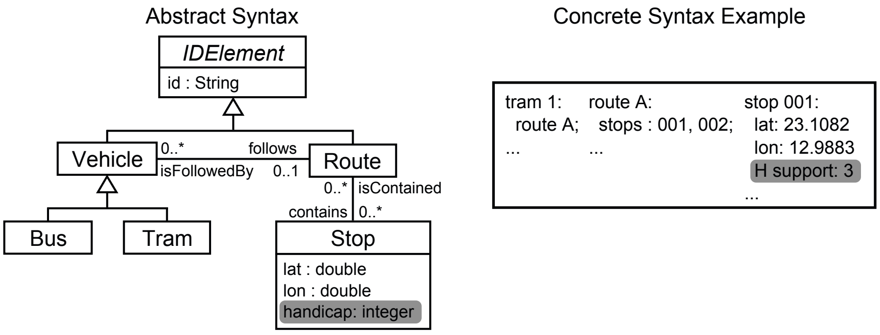


Fig. 1. Abstract syntax and an example of concrete syntax of the *Transport* DSML (grey-filled boxes represent elements added after the collaboration)

The development of a DSML consists in five different phases [4]: decision, analysis, design, implementation and deployment. The first three phases are mainly focused on the DSML definition whereas the implementation phase is aimed at developing the tooling support (i.e., modeling environment, parser,...) for the DSML. Clearly, the community around the language is a key element in the process. We call *community* the group of people involved with the DSML under development, which includes both technical level users (i.e., language developers) and domain expert users (i.e., end-users of the language), where both categories can overlap, especially when the DSML is a technical DSML.

As a running example, imagine the development of a DSML for managing the transportation service of a city. Typically, the domain expert users are only heavily involved at the very beginning and very end of the process. Assuming this is also the case for our transportation DSL, during the analysis phase, the developers will study the transportation domain with the help of the city hall members in charge of the city transportation policies and decide that it should include concepts such as `Vehicle` (i.e., `Bus` or `Tram`), and `Route`, which is composed by `Stops`. The developers therefore design and later implement the tooling of the language, thus coming up with a textual DSML whose abstract syntax and an example of the concrete syntax are shown in Figure 1 (all elements except the ones included in grey-filled boxes). Once the language is developed, end-users can play with it and check whether it fits their needs. Quite often, if the end-users only provided the initial input but did not closely follow how that was interpreted during the language design, they will detect problems in the modeling environment (e.g., missing concepts, wrong notation, etc.) that will trigger a new (and costly) iteration to modify the language and recreate all the associated tools. For instance, end-users could detect that the language lacks of a construct to represent whether a `Stop` is adapted for handicapped people.

Therefore, our aim is to incorporate the community collaboration aspect into all DSML definition phases, making the early phases of the process more participative and promoting the early detection of possible bugs or problems. The resulting collaborative process is summarized in Figure 2. Once there is an agreement to create the language, community defines the collaboration strategy to make decisions (e.g., how

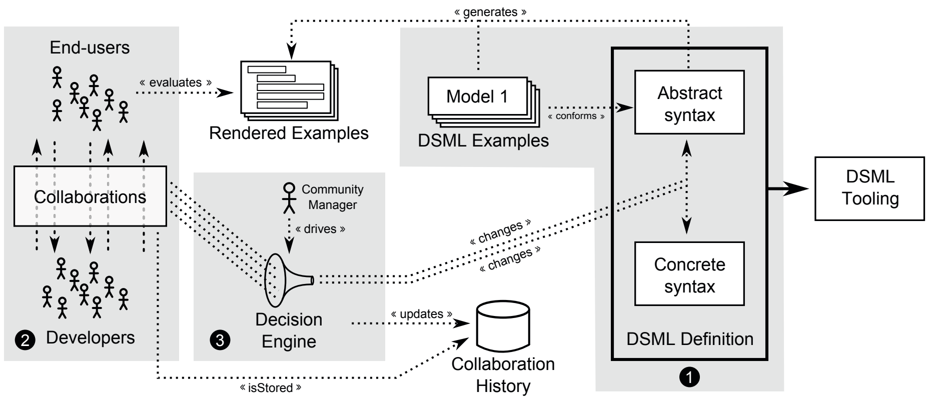
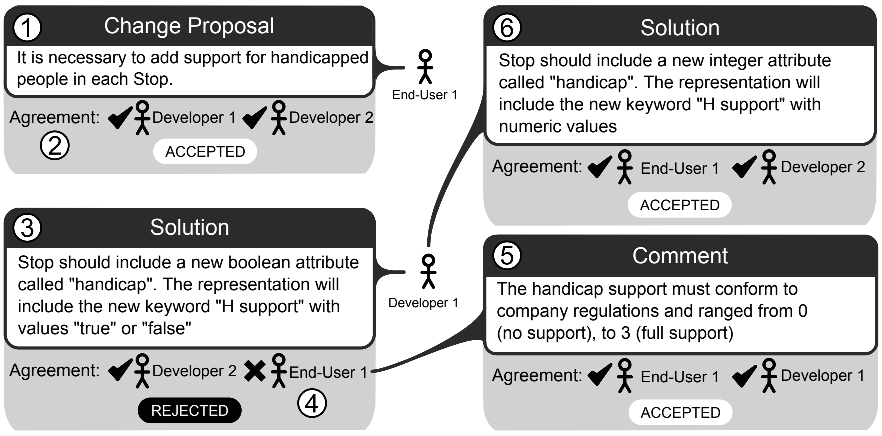


Fig. 2. Collaborative development of DSMLs

can vote, number of votes to reach an agreement, etc.). Next, language developers get the requirements from the end-users to create a preliminary version of the language to kickstart the actual collaboration process (step 1). This first version should include at least a partial abstract syntax but could also include a first concrete syntax draft (see *DSML Definition*). An initial set of sample models conforming to the DSML are also defined to facilitate an example-based discussion, usually easier for non-technical users. These sample models are rendered according to the current concrete syntax definition (see *Rendered Examples*). It is worth noting that the rendering is done on-the-fly without the burden of generating the DSML tooling since we are just showing the snapshots of the models to discuss the notation, not actually providing at this point a full modeling environment.

Now the community starts working together in order to shape the language (step 2). Community members can propose ideas or changes to the DSML, e.g., they can ask for modifications on how some concepts should be represented (both at the abstract and concrete syntax levels). These change proposals are shared in the community, who can also suggest and discuss how to improve the change proposals themselves. All community members can also suggest solutions for the requested changes and give their opinion on the solutions presented by others. At any time, rendering the sample models with the latest proposals gives members an idea of how a proposal will evolve the language (if accepted). All these proposals and solutions (see *Collaborations*) are eventually accepted or rejected.

Acceptance/rejection depends on whether the community reaches an agreement regarding the proposal/solution. For that, community members can vote (step 3). A decision engine (see *Decision Engine*) then takes these votes into account to calculate which collaborations are accepted/rejected by the community. The engine could follow an automatic process (according to the collaboration strategy defined at the beginning of the process) but a specific role of *community manager* could also be assigned to a member/s to consolidate the proposals and get a consensus on conflicting opinions (e.g., when there is no agreement between technical and business considerations). Once an



1 Change Proposal
It is necessary to add support for handicapped people in each Stop.
Agreement: Developer 1 Developer 2
2
ACCEPTED
End-User 1
6 Solution
Stop should include a new integer attribute called "handicap". The representation will include the new keyword "H support" with numeric values
Agreement: End-User 1 Developer 2
ACCEPTED
3 Solution
Stop should include a new boolean attribute called "handicap". The representation will include the new keyword "H support" with values "true" or "false"
Agreement: Developer 2 End-User 1
REJECTED
4
Developer 1
5 Comment
The handicap support must conform to company regulations and ranged from 0 (no support), to 3 (full support)
Agreement: End-User 1 Developer 1
ACCEPTED

Fig. 3. Example of collaboration in the *Transport* DSML

agreement is reached, the contents of the solution are incorporated into the language, thus creating a new version. The process keeps iterating until no more changes are proposed. Note that these changes on the language may also have an impact on the model examples which may need to be updated to the new language definition.

At the end of the collaboration, the final DSML definition is used to implement the DSML tooling (see *DSML Tooling*) with the confidence that it has been validated by the community. Note that even whether the language does not comply to commonly applied quality patterns (e.g., graphical vs. textual, redundant elements in the DSML, etc.), developers can be sure that fulfills the end-users' needs. Moreover, all aspects of the collaboration are recorded (see *Collaboration History*), thus keeping track of every interaction and change performed in the language. Therefore, at any moment, this traceability information can be queried to discover the rationale behind the elements of the language (e.g., the argumentation provided for its acceptance).

To illustrate our approach, the development of the *Transport* DSML mentioned above could have been included the following collaboration scenario. The developers start designing a very first version of the language. Then, the collaboration begins and a community member detects the need of expressing the support for handicapped people when checking some language snapshots. Since now we are still in the definition phase the community has the chance to discuss the best way to adapt the language to support this new handicapped information. The member that identified a problem would create a change proposal. If the change is deemed as important by the community, other members could propose a solution/s to adapt the language. As an example, Figure 3 graphically depicts a possible collaboration scenario assuming a small community of one end-user and two developers. Each collaboration is represented as a bubble and each step has been numbered. In the Figure, *End-User 1* proposes a language change (step 1), which is accepted by the community (step 2), and then *Developer 1* specifies a solution (step 3). The solution is rejected by *End-User 1*, including also the explanation of the rejection (step 4). As the rejection is accepted (step 5), the *Developer 1* redefines the solution, which is eventually accepted (step 6) and the changes are then incorporated into the language. The resulting changes in the abstract and concrete syntaxes are shown in grey-filled boxes in Figure 1a. Clearly, it is important to make this collaboration iterations as quick as possible and with the participation of all types of community members. Moreover, the discussion information itself must be preserved to justify the rationale behind each language evolution, from which design decisions can be derived.

3 Collaboro DSML

Our proposal for enabling the collaborative definition of DSMLs is built on top of the *Collaboro* DSML, a DSML for modeling the collaborations that arise in a community working towards the development of a DSML for that community. In the next sections we will describe how Collaboro makes the collaboration feasible by (1) enabling the discussion about DSML elements, (2) providing the metaclasses for representing collaborations and (3) giving support to the decision-making process regarding the changes to be incorporated into the DSML based on the results of the collaboration so far.

3.1 Representing the Elements of a DSML

To be able to discuss about changes on the DSML to-be, we must be able to represent its elements both the abstract syntax (i.e., the concepts of the DSML) and the concrete syntax (the notation to represent those concepts) elements. The abstract syntax is commonly defined by means of a metamodel written using a metamodelling language (e.g., MOF or Ecore). Metamodelling languages normally offer a limited set of concepts to be used when creating DSML metamodels (like types, relationship or hierarchy). A DSML metamodel is then defined as an instantiation of this metamodeling concepts. Figure 4a shows an excerpt of the well-known Ecore metamodelling language.

Regarding the concrete syntax, since the notation of a DSML is also domain-specific, to promote the discussion we need to be able to explicitly represent the elements defining the notation of a DSML. Thus, community members will have the freedom to create a notation specially adapted to their domain, thus avoiding coupling with other existing notations (e.g., Java-based textual languages or UML-like diagrams). Given that nowadays there is no technology-independent metamodel to model the concrete syntax of a DSML, we have defined our own metamodel for concrete syntaxes. Figure 4b shows an excerpt of the core elements of this notation metamodel. As can be seen, the metamodel is far from exhaustive but it suffices to discuss about the concrete syntax elements most commonly used in the definition of graphical, textual or hybrid concrete syntaxes. Note that with this metamodel, it is possible to describe how to represent each language concept, thus facilitating keeping track of language changes.

Concrete syntax elements are classified following the `NotationElement` hierarchy, which includes graphical elements (`GraphicalElement` metaclass), textual elements (`TextualElement` metaclass), composite elements (`Composite` metaclass) and references to the concrete syntax of other abstract elements (`SyntaxOf` metaclass) to be used in composite patterns. The main graphical constructs are provided by the `GraphicalElement` hierarchy, which allows referring to external pictures

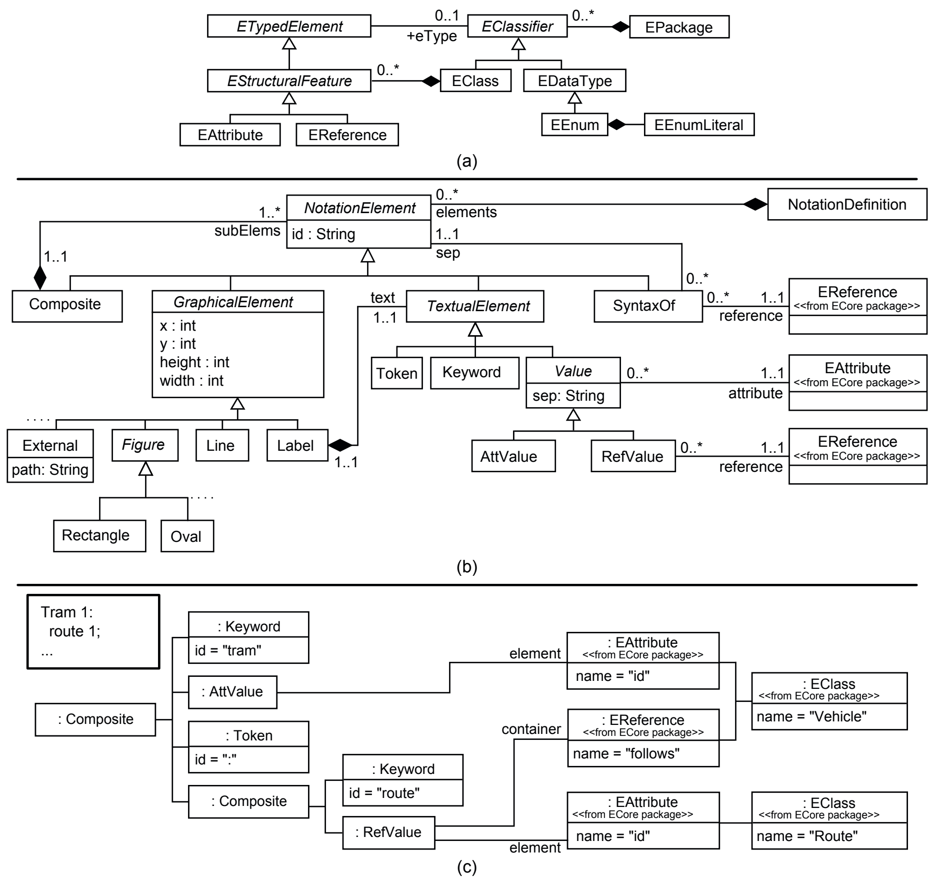


Fig. 4. (a) Excerpt of the Ecore metamodel. (b) Excerpt of the Notation metamodel. (c) Notation model for the textual representation of the metaclass `Tram` of the *Transport* DSML.

(`External` metaclass), building figures (see `Figure` hierarchy), lines (`Line` metaclass) and labels for the DSML elements. A label (`Label` metaclass) serves as a container for a textual element. Textual elements can be defined with the `TextualElement` hierarchy, which includes tokens, keywords and values directly taken from the abstract syntax elements expressed in a textual form (`Value` metaclass). It is possible to obtain the textual representation from either an attribute (`AttValue` metaclass) by specifying the attribute to be queried (`attribute` reference), or a reference (`RefValue` metaclass) by specifying both the reference (`reference` reference) and the attribute of the referred element to be used (`attribute` reference). The attribute `sep` of the `Value` metaclass allows defining the separator for multivalued elements. The `Composite` element can be used to define complex concrete syntax structures, allowing both graphical and textual composites but also hybrids. Finally, the `SyntaxOf` metaclass allows referencing to already specified concrete syntax definitions of abstract syntax elements, thus allowing modularization and composition. The `reference` reference of the

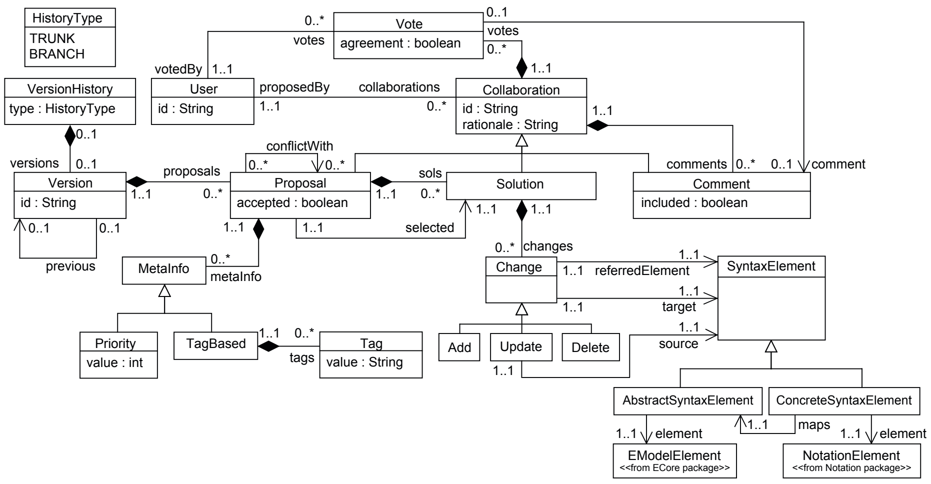


Fig. 5. Core elements of the collaboration metamodel

SyntaxOf metaclass specifies the reference to be queried to obtain the set of elements whereas the sep reference indicates the separator between elements.

As example of the notation metamodel, Figure 4c shows the notation model for the textual representation of the metaclass Tram of the *Transport* DSML. Note that AttValue and RefValue metaclass instances are referring to elements from the abstract syntax metamodel.

3.2 Representing the Collaborations

The third metamodel required in the Collaboro process focuses on representing the collaborations that annotate/modify the DSML elements described before. This collaboration metamodel, which is shown in Figure 5, allows representing both static (e.g., change proposals) and dynamic (e.g., voting) aspects of the collaboration. A preliminary version of this metamodel was presented in [9] but, among other limitations, only supported the collaborative design of abstract syntaxes.

Static Part. Similarly to how version control systems track source code, Collaboro also allows representing different versions of a DSML. The VersionHistory metaclass represents the set of versions (Version metaclass) through which the collaboration evolves. There is always a main version history set as *trunk* (type attribute in VersionHistory metaclass), which keeps the baseline of the collaborations about the language under development. Other version histories (similar to branches) can be forked when it is necessary to isolate the collaboration about concrete parts of the language. Different version histories can be merged into a new one (or the *trunk*).

Language evolution is the consequence of collaborations (Collaboration metaclass). Collaboro supports three types of collaborations: change proposals (Proposal metaclass), solutions proposals (Solution metaclass) and comments (Comment metaclass). A collaboration is proposed by a user (proposedBy reference) and includes an explanation (rationale attribute).

A change proposal describes which language feature should be modified and contains some meta information (e.g., priority or tags). Change proposals are linked to the set of solutions proposed by the community to be discussed for agreement. It is also possible to specify possible conflicts between similar proposals (e.g., the acceptance of one proposal can involve rejecting others) with the `conflictWith` reference.

Solution proposals are the answer to change proposals and describe how the language should be modified to incorporate the new features. Each solution definition involves a set of add/update/delete changes on the elements of the DSML (`Change` hierarchy). `Change` links the collaboration metamodel with the DSML under discussion (`SyntaxElement` metaclass), which can refer to the abstract syntax (`AbstractSyn taxElement` metaclass) or the concrete syntax (`ConcreteSyntaxElement` metaclass). The latter links (`maps` reference) to the abstract element to which the notation is defined. Both `AbstractSyntaxElement` and `ConcreteSyntaxElement` have a reference linking to the element which is being changed (`element` reference). Changes in the abstract syntax are expressed in terms of the metamodelling language (i.e., `EMode lElement` elements, which is the interface implemented by every element in the Ecore metamodel) while changes in the concrete syntax are expressed in terms of elements conforming to the notation metamodel presented before.

The metaclass `Change` has a reference to the container element affected by the change (`referredElement` reference) and the element to change (`target` reference). Thereby, in the case of `Add` and `Delete` metaclasses, `referredElement` refers to the element to which we want to add/delete a "child" element whereas `target` refers to the actual element to be added/deleted. In the case of the `Update` metaclass, `referredElement` refers to the element which contains the element to be updated (e.g., a metaclass) whereas `target` refers to the new version of the element being updated (e.g., a new version for an attribute). The additional `source` attribute indicates the element to be updated (e.g., the attribute which is being updated).

Dynamic Part. During the process, community members vote collaboration elements, thus allowing to reach agreements. Votes (`Vote` metaclass) indicate whether the user (`votedBy` reference) agrees or not with a collaboration (`agreement` attribute). A vote against a collaboration usually includes a comment explaining the reason of the disagreement (`comment` reference of `Vote` metaclass). This comment can then be voted itself and if it is accepted by the community, the proponent of the voted proposal/solution should take such comment into account (the `included` attribute of `Comment` metaclass records this fact).

The acceptance of a proposal means that the community agrees that the requested change is necessary (`accepted` attribute). For each proposal we can have several solutions but in the end one of them will be selected (`selected` reference of the `Proposal` metaclass) and its changes applied to the DSML definition. Part of this data (like the `accepted` and `selected` properties) is automatically filled by the decision engine analyzing and resolving the collaboration.

3.3 Decision Engine

As explained in Section 2, community votes are used to decide which collaborations are accepted and must be incorporated into the language. Collaboration models include

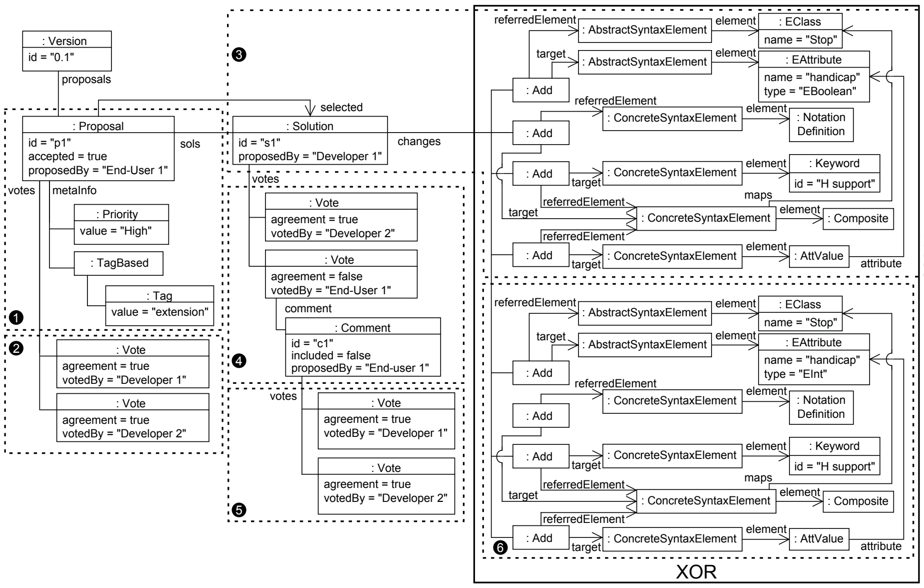


Fig. 6. The collaboration model representing the collaborations arisen in the *Transport* DSML

all the necessary collaboration information, thus allowing the automation of the decision process (i.e., approval of change proposals and selection of solutions). A decision engine can therefore apply resolution strategies (e.g., unanimous agreement, majority agreement, etc) to deduce (and apply) the collaborations accepted by the community. As commented before, most times it is necessary to have the role of the community manager to trigger the decision process and solve possible decision locks.

The application of resolution strategies could be implemented as in-place model-to-model transformations for collaboration models for instance using graph transformation rules enforcing the agreed decision policies and updating the involved models (i.e., collaboration model, DSML metamodel and notation model) accordingly.

3.4 Example

To illustrate the proposed infrastructure, we show in Figure 6 the collaboration model which would be obtained as a result of the example discussed in Section 2. The figure is divided in several sections according to the collaboration steps enumerated previously. For the sake of clarity, the references to `User` metaclass instances have been represented as string-based attributes and the `rationale` attribute is not shown.

Section 1 of Figure 6 shows the collaboration model just after *End-User 1* makes the request. It includes a new `Proposal` instance whose `id` attribute is `p1`. The proposal meta-information specifies that such proposal is `High` priority and has the tag `extension`. Once the proposal has been created, the community can vote for/against it as well as add comments and solutions. In this case, the proposal is voted positively by the rest of the users and therefore accepted (see the `Vote` instances referred by the proposal in the section 2 of Figure 6). Then, a new solution is proposed by *Developer*

1 (see the `Solution` instance in section 3 of Figure 6), which involves enriching the `Stop` metaclass with a boolean-based attribute in addition to define the concrete syntax.

However, this solution is not accepted by all the community members: when voting such solution, *End-User 1* does not agree and explains his disagreement with the corresponding comment (see section 4 of Figure 6). Since the comment is accepted (see section 5 of Figure 6), *Developer 1* decides to update the solution to incorporate the community recommendations (see section 6 of Figure 6). Note that the elements describing the model changes in sections 3 and 6 are mutually exclusive (i.e., section 6 is an evolution of section 3 once the community agrees that the comment from *End-User 1* must be taken into account). Moreover, the attribute `included` of the `Comment` element in section 4 of Figure 6 will be activated as a consequence of the solution update.

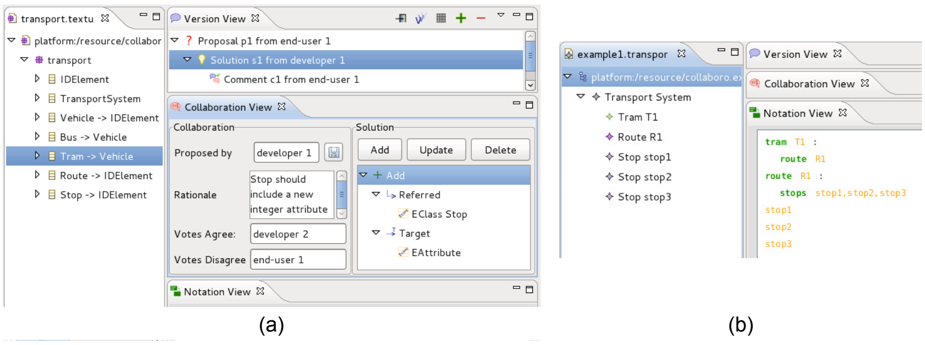
Once everybody agrees on the improved solution, it is selected as the final solution for the proposal (the `selected` reference is initialized with the `Solution` instance). Now the development team can modify the DSML tooling knowing that the community needs the language to be changed and agrees on how the change must be done. Moreover, the rationale of the change will be tracked by the collaboration model (from which an explanation in natural language could be generated, if needed), which will allow community members to know why the `Stop` metaclass was changed.

4 Tool Support

We have developed an Eclipse plugin[4] implementing the Collaboro process and DSML. Current version works with the EMF framework (the standard *de facto* modeling framework nowadays) and includes a decision engine implementing a total agreement strategy to infer community agreements from the voting information as well as a notation generator to enable the lightweight creation of SVG images from notation models to help users "see" how the notation they are discussing will look like when used to define models with that DSL. To support concurrent collaboration the tool uses the CDO[5] model repository.

The plugin provides a set of new Eclipse views and editors to facilitate the collaboration, which can be considered a kind of concrete syntax of Collaboro itself for non-expert users. Figure 7a includes a snapshot of the environment showing the last step of the collaboration described in Section 3.4. In particular, the *Version view* lists the collaboration elements (i.e., proposals, solutions and comments) of the current version of the collaboration model. The *Collaboration View* shows the detailed information of the selected collaboration element in the Version view and a tree-based editor to indicate the changes to discuss for that element, as shown in Figure 7a. Finally, the *Notation view* uses the notation generator to render a full example model of the language. For instance, the Notation view in Figure 7b shows the notation for an example model, which allowed detecting the missing attribute regarding handicapped support.

[4] `http://code.google.com/a/eclipselabs.org/p/collaboro`
[5] `http://www.eclipse.org/cdo`



Proposed by: developer 1
Rationale: Stop should include a new integer attribute
Votes Agree: developer 2
Votes Disagree: end-user 1

```
tram T1 :
  route R1
route R1 :
  stops stop1,stop2,stop3
stop1
stop2
stop3
```

Fig. 7. (a) Snapshot of the Collaboro Eclipse plugin. (b) Collaboro Eclipse plugin with the *Notation view* rendering the concrete syntax for a model.

5 Case Study

We have used Collaboro in the development of a new DSML for MoDisco[6], an Eclipse project aimed at defining a group of tools for Model-Driven Reverse Engineering (MDRE) processes. The goal of this new DSML is facilitating the development of MDRE workflows that chain several atomic reverse engineering tasks to extract the model/s of a running system. At the moment, the only way to define a MDRE workflow is by using an interactive wizard. MoDisco users have been asking for a specific language to do the same in a more direct way, i.e., without having to go through the wizard.

Some years ago an initial attempt to create such language was finally abandoned but, to simplify the case study, we reused the metamodel that was proposed at the time to kickstart the process. The initial version of the workflow metamodel is shown in Figure 8 (elements in black). A workflow element (`Element` metaclass) is identified by its name (`name` attribute) and has a type (`type` attribute) and an index (`index` attribute) specifying the order in which such element will be executed. There are two types of elements: workflows (`Workflow` metaclass), which represents a workflow itself; and works (`Work` metaclass), which represents individual tasks to execute. The `Workflow` metaclass inherits from the `ExportInfos` metaclass, that can be used to indicate additional metadata. Each work parameter (`WorkParameter` metaclass) includes a name (`name` attribute), a type (`type` attribute), a direction (`direction` attribute), whether they are mandatory (`required` attribute) and a description (`description` attribute). Parameters can have a value assigned (`WorkParameterValue` hierarchy).

Five researchers of the team followed our collaborative process to complete/improve the abstract syntax of the DSML and create from scratch a concrete syntax for it. Two of the members were part of the MoDisco development team so they took the role of developers in the process while the other three were only users of MoDisco so they adopted more the role of end-user in the process. One of the members was in a different country during the collaboration so only asynchronous communication was possible.

[6] `http://eclipse.org/modisco`

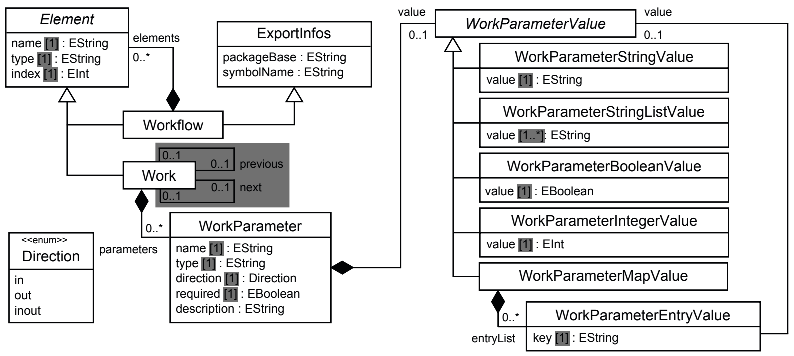


Fig. 8. Abstract syntax of the MDRE *workflow* language (grey-filled boxes show the changes of the resulting abstract syntax)

Table 1. Change proposals and solutions for the two versions of the MDRE *workflow* language

	Change proposal rationale	**Solution rationale**
First version	The information about the Eclipse plugin represented in the `ExportInfos` metaclass can be removed.	`ExportInfos` metaclass was removed
	Most attributes have lower multiplicity equal to 0. If the attributes are mandatory it should be equal to 1.	Lower cardinality of all the attributes has been revised and set to 1 when required.
	`Work` metaclass should have references to both the previous and next `Elements` in the workflow	`Work` metaclass incorporates two new references called `previous/next` and referring to the previous/next `Elements`, respectively
Second version	The textual concrete syntax should follow a block-based syntax	Each metaclass will be represented by the same named keyword and then the textual representation of its attributes
	Keywords will be represented in red and the rest in black	The color for representing keywords was updated
	The `WorkParameter` syntax should indicate the direction	The syntax will include the value of the `direction` enumerated attribute
	The `Work` syntax should includes the name of the previous/next `Element`'s name	The syntax will include the name of the `Element` after the `Work`'s name definition

The collaboration took two weeks and resulted in two new versions of the MDRE workflow language released. The first version was mainly focused on the polishment of the abstract syntax whereas the second one paid more attention to the concrete syntax (this was not enforced by us but it came out naturally). Regarding the collaborations arisen, the first version involved 7 change proposals (3 were accepted) and 5 comments; whereas the second one included 7 proposals (4 were accepted) and 14 comments. Table 1 summarizes the main proposals/solutions accepted [7]. The collaboration regarding the abstract syntax involved the changes shown in grey-filled boxes in Figure 8 (note that the `ExportInfos` metaclass was removed). Regarding the concrete syntax, the community preferred a textual-based notation[8] and mainly commented on which keywords or style should be used.

[7] The complete collaboration model can be downloaded from `http://goo.gl/RxJ3o`
[8] The notation model can be downloaded from `http://goo.gl/wRoCH`

5.1 Lessons Learned

The case study provided us with some useful insights on the collaboro process that since then have been already integrated in Collaboro. For instance, it turned out that conflicting proposals were frequent so we added a conflicting relationship information explicitly in the collaboration metamodel so that once one of them was accepted we could automatically shut down the related ones. We also noted an intensive use of comments (easier to add) in comparison with proposals and solutions. This, linked with the discussions of what constituted a new version and when to end the discussions (e.g., if there was a unanimity but not everybody had voted, should we wait for that person? for how long?) helped us to realize the need of an explicit community manager figure in charge of making sure the collaboration is always fluid and there are no bottlenecks or deadlocks. Moreover, concurrent access to the models (supported by the tool) turned out to be a must as well since most of the time collaborations overlapped.

The notation view allowed the participants to quickly validate the concrete syntax. This is specially important since for non-technical users is easier to discuss at the concrete syntax level than at the abstract level. However, we are aware that the process of defining notation models (and the corresponding example models to be rendered) implies a considerable workload. Thus, we plan to incorporate support for a "change by example" approach where end-users can suggest changes by providing example models (possibly inconsistent with the current DSML version) of how they would like to represent certain scenarios and ask the developers to adapt the DSML to render them valid, in a similar way as proposed in [10].

The only complain we got was regarding the limited support for voting. They would have preferred more options instead of just a boolean yes/no option. Note that this would have a non negligible impact on the decision algorithms that would need to be adapted to consider the new voting options. This is a clear trade-off to be analyzed individually for each DSML.

All in all, we believe the collaboration was a success since all participants were happy about the final DSML and agreed that at least now they could actually use it (instead of the original one who was never adopted). Obviously, these results cannot be generalized since the number and profile of the users (all of them familiar with MDE technologies) is a clear threat to the validity of the experiment which would need to be replicated to gather more information about the benefits and challenges of using Collaboro.

6 Related Work

End-user involvement is a core feature of several software development methods (such as agile-based ones). The concept of *community-driven development* in the development of a software product was introduced in [11] and other authors have studied this collaboration as part of the requirement elicitation [12] and modeling phases of the software [2,13,14,15], but neither of them focuses on the DSML language design process nor they present the collaboration as a process of discussion and argumentation from the beginning to the end of the development process. End-user participation is also the core of user-centered design [16], initially focused on the design of user interfaces but lately applied to other domains (e.g., agile methodologies [17] or web development

[18]). Again, none of these approaches can be directly applied to the specification of a DSML. Nevertheless, ideas from these papers have indeed influenced the Collaboro process.

Regarding specific approaches around collaboration in DSML development, [5,19,10] propose to derive a first DSML definition by means of user demonstrations, where example models provided by end-users are analyzed to derive the metamodel of the language. However, these approaches do not include any discussion phase nor validation of the generated metamodel with the end-users. In this sense, our approaches could complement each other, theirs could be used to create an initial metamodel from which start the refinement process based on the discussions among the different users.

Subsets of our proposal can also be linked to: i) specific tools for model versioning (e.g., AMOR repository[9] and [20]) that have already proposed a taxonomy of metamodel changes, ii) online-collaboration ([21,22]) promoting synchronous collaboration among developers, iii) metamodel-centric language definition approaches ([23,24]) where the concrete syntax is considered at the same level as the abstract one and iv) collaboration protocols [25]. In all cases, Collaboro extends the contributions of those tools with explicit collaboration and justification constructs, and provides as well the possibility of offline collaborations and a more formal representation of the interactions (e.g., voting system, explicit argumentation and rationale, traceability).

Finally, the representation of the collaboration rationale is related to the area of requirements negotiation, argumentation and justification approaches such as [7]. The decision algorithms proposed in those works could be integrated in our decision engine.

7 Conclusion

We have presented Collaboro, a DSML to enable the participation of all members of a community in the specification of a new domain-specific language. Collaboro allows representing (and tracking) language change proposals, solutions and comments for both the abstract and concrete syntaxes of the language. This information can then be used to justify the design decisions taken during the language definition.

As further work, we are working on how to support the collaborative definition of the well-formed rules (e.g., OCL constraints) for the DSML under development and the integration of "model by example" approaches where users can propose new solutions by using example models they would like to be able to define (or not) with the DSML. We also plan to study how to encourage end-user participation (e.g., by applying gamification techniques) in the language definition process as well as how to provide some assistance along the way, for instance, by providing a notation recommender component implementing well-known guidelines for concrete syntax creation (like [26,27]) that could help users choose the best notation option for a given element/s.

References

1. Hatton, L., Genuchten, M.V.: Early design decisions. IEEE Softw. 29(1), 87–89 (2012)
2. Hildenbrand, T., Rothlauf, F., Geisser, M., Heinzl, A., Kude, T.: Approaches to collaborative software development. In: FOSE Conf., pp. 523–528. IEEE (2008)

[9] `http://www.modelversioning.org`

3. Kelly, S., Pohjonen, R.: Worst practices for domain-specific modeling. IEEE Softw. 26(4), 22–29 (2009)
4. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37, 316–344 (2005)
5. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: MiSE Workshop, pp. 29–35. IEEE (2012)
6. Cabot, J., Wilson, G.: A Survey of Web-Based Software Project Portals. Dr. Dobbs (2009)
7. Jureta, I., Faulkner, S., Schobbens, P.Y.: Clear justification of modeling decisions for goal-oriented requirements engineering. Requir. Eng. 13, 87–115 (2008)
8. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley (2008)
9. Cánovas Izquierdo, J.L., Cabot, J.: Community-driven language development. In: MiSE Workshop, pp. 22–28. IEEE (2012)
10. Sánchez-Cuadrado, J., de Lara, J., Guerra, E.: Bottom-up Meta-Modelling: an Interactive Approach. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 3–19. Springer, Heidelberg (2012)
11. Hess, J., Offenberg, S., Pipek, V.: Community driven development as participation?: involving user communities in a software design process. In: PD Conf., pp. 31–40 (2008)
12. Mylopoulos, J., Chung, L., Yu, E.: From Object-Oriented to Goal-Oriented Requirements Analysis. Commun. ACM 42(1), 31–37 (1999)
13. Lanubile, F., Ebert, C., Prikladnicki, R., Vizcaino, A.: Collaboration tools for global software engineering. IEEE Softw. 27(2), 52–55 (2010)
14. Whitehead, J.: Collaboration in software engineering: A roadmap. In: FOSE Conf., pp. 214–225. IEEE (2007)
15. Rittgen, P.: COMA: A tool for collaborative modeling. In: CAiSE Forum, pp. 61–64 (2008)
16. Norman, D.A., Draper, S.W.: User Centered System Design: New Perspectives on Human-computer Interaction. Erlbaum, Hillsdale (1986)
17. Hussain, Z., Slany, W., Holzinger, A.: Current state of agile user-centered design: A survey. In: Holzinger, A., Miesenberger, K. (eds.) USAB 2009. LNCS, vol. 5889, pp. 416–427. Springer, Heidelberg (2009)
18. De Troyer, O., Leune, C.: WSDM: a user centered design method for Web sites. Computer Networks and ISDN Systems 30(1-7), 85–94 (1998)
19. Kuhrmann, M.: User assistance during domain-specific language design. In: FlexiTools Workshop (2011)
20. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Web Inf. Syst. 5(3), 271–304 (2009)
21. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: We can work it out: Collaborative conflict resolution in model versioning. In: CSCW Conf., pp. 207–214 (2009)
22. Gallardo, J., Bravo, C., Redondo, M.A.: A model-driven development method for collaborative modeling tools. Netw. and Comput. Appl. (2011)
23. Scheidgen, M.: Textual modelling embedded into graphical modelling. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 153–168. Springer, Heidelberg (2008)
24. Prinz, A., Scheidgen, M., Tveit, M.S.: A model-based standard for SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, pp. 1–18. Springer, Heidelberg (2007)
25. Gallardo, J., Bravo, C., Redondo, M.A., de Lara, J.: Modeling collaboration protocols for collaborative modeling tools: Experiences and applications. Vis. Lang. & Comput., 1–14 (2012)
26. Rittgen, P.: Negotiating models. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 561–573. Springer, Heidelberg (2007)
27. Moody, D.L.: The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. IEEE Trans. Soft. Eng. 35(6), 756–779 (2009)