A peer-reviewed version of this preprint was published in PeerJ on 24 October 2016.

View the peer-reviewed version (peerj.com/articles/cs-84), which is the preferred citable publication unless you specifically need to cite this preprint.

Cánovas Izquierdo JL, Cabot J. 2016. Collaboro: a collaborative (meta) modeling tool. PeerJ Computer Science 2:e84
https://doi.org/10.7717/peerj-cs.84

Collaboro: A collaborative (Meta) modeling tool

Javier Luis Cánovas Izquierdo, Jordi Cabot

Software development processes are collaborative in nature. Neglecting the key role of end-users leads to software unlikely to 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. Collaboro also incorporates a metric-based recommender system to help community members to define high-quality notations for the DSMLs. We also show how Collaboro can be used at the model-level to facilitate the collaborative specification of software models.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2046v1 | CC-BY 4.0 Open Access | rec: 17 May 2016, publ: 17 May 2016

Collaboro: A Collaborative (Meta) Modeling Tool

Javier Luis Cánovas Izquierdo[*1] **and Jordi Cabot**[2]

[1,2]**UOC, Barcelona, Spain**
[2]**ICREA, Barcelona, Spain**

ABSTRACT

Software development processes are collaborative in nature. Neglecting the key role of end-users leads to software unlikely to 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. Collaboro also incorporates a metric-based recommender system to help community members to define high-quality notations for the DSMLs. We also show how Collaboro can be used at the model-level to facilitate the collaborative specification of software models.

Keywords: Collaborative Development, Domain-Specific Languages, Model-Driven Development

INTRODUCTION

Collaboration is key in software development, it promotes a continual validation of the software to be build (Hildenbrand et al., 2008), thus guaranteeing that the final software will satisfy the users' needs. Furthermore, the sooner the end-users participate in the development life-cycle, the better, as several works claim (Hatton and van Genuchten, 2012; Rooksby and Ikeya, 2012; Dullemond et al., 2014). When the software artefacts being developed target 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 performing (meta) modeling tasks.

On the one hand, end-users are key when defining a Domain-Specific Modeling Language (DSML), a modeling language specifically designed to perform a task in a certain domain (Sánchez Cuadrado and García Molina, 2007). 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 Grundy et al. (2013). Therefore, the role of domain experts during the DSML specification is vital, as noted by several authors Kelly and Pohjonen (2009); Mernik et al. (2005); Völter (2011); Barišić et al. (2012). Unfortunately, nowadays, participation 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 (also scarcely done as reported in Gabriel et al. (2010)), which requires the development of fully functional language toolsets (including a model editor, a parser, etc.) Mernik et al. (2005); Cho et al. (2012). This long iteration cycle is a time-consuming and repetitive task that hinders the process performance Kelly and Pohjonen (2009) since end-users must wait until the end to see if designers correctly understood all the intricacies of the domain. On the other hand, those same end-users will then employ that modeling language (or any general-purpose modeling language like UML) to specify the systems to be built. Collaboration here is also key in order to enable the participation of several problem experts in the process. Recently, modeling tools have been increasingly enabling the collaborative development of models defined by GPLs and DSLs. However,

[*]Corresponding Author. Address: IN3 - UOC. Av. Carl Friedrich Gauss, 5. Building B3. 08860 Castelldefels. Email: jcanovasi@uoc.edu

their support for asynchronous collaboration is still limited, specially when it comes to the traceability and justification of modeling decisions.

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 Cabot and Wilson (2009). 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 or using modeling languages, where a special support to discuss at language element level (i.e., domain concepts and notation symbols) is required to address the challenges previously described and therefore promote the participation of end-users. As mentioned above, a second major problem shared by current solutions is the lack of traceability of the design decisions. The rationale behind decisions made during the language/model specification are implicit so it is not possible to understand or justify why, for instance, a certain element of the language was created with that specific syntax or given that particular type. This hampers the future evolution of the language/model.

In order to alleviate these shortcomings, we define a DSML called *Collaboro*, which enables the involvement of the community (i.e., end-users and developers) in the development of (meta) modeling tasks. 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. Also, it can be easily adapted to enable the collaborative definition of models. Thus, each community member has the chance to request changes, propose solutions and give an opinion (and vote) on those from others. We believe this discussion enriches the language definition and usage significantly, and ensures that the end result satisfies as much as possible the expectations of the end-users. Moreover, the explicit recording of these interactions provides plenty of valuable information to explain the language evolution and justify all design decisions behind it, as also proposed in requirements engineering Jureta et al. (2008). Together with the Collaboro DSML, we provide the tooling infrastructure and process guidance required to apply Collaboro in practice.

The first version of Collaboro, which supported the collaborative development of textual DSMLs in an Eclipse-based enviroment, was presented in a previous work by Cánovas Izquierdo and Cabot (2013). Since then, the approach has evolved to include new features such as: (1) support for the complete collaborative development of graphical DSMLs, (2) a new architecture which includes a web-based front-end, thus promoting usability and participation for end-users; and (3) a metric-based recommender system, which checks the DSMLs under development to spot possible issues according to quality metrics for both the domain and the notation (relying on Moody's cognitive framework Moody (2009)). In addition, the development of new features along with the experience gained in using the language in several cases studies allowed us to improve its general expressiveness and usability and realize the benefits of Collaboro also at the model level.

Paper structure. The first two sections describe the proposal and approach to develop DSML collaborative. The following section shows then how our approach could be easily adapted to use any modeling language to model collaboratively. Next, the implemented tool and a case study are described. Finally, we review the related work and draw some conclusions and future work.

COLLABORATIVE (META) MODELING

While collaboration is crucial in both defining modeling languages and then using them to model concrete systems, the collaborative aspects of language development are more challenging and less studied since collaboration must cover both the definition of a new notation for the language and the specification of the language primitives themselves. Therefore, we will present first Collaboro in the context of collaborative language development and later its adaptation to cover the simpler modeling scenario. A running example, also introduced in this section, will help to illustrate the main concepts of such collaborations.

A DSML is defined through three main components Kleppe (2008): 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. Metamodeling techniques are

[1] http://trac.edgewall.org/
[2] http://www.atlassian.com/es/software/jira/overview

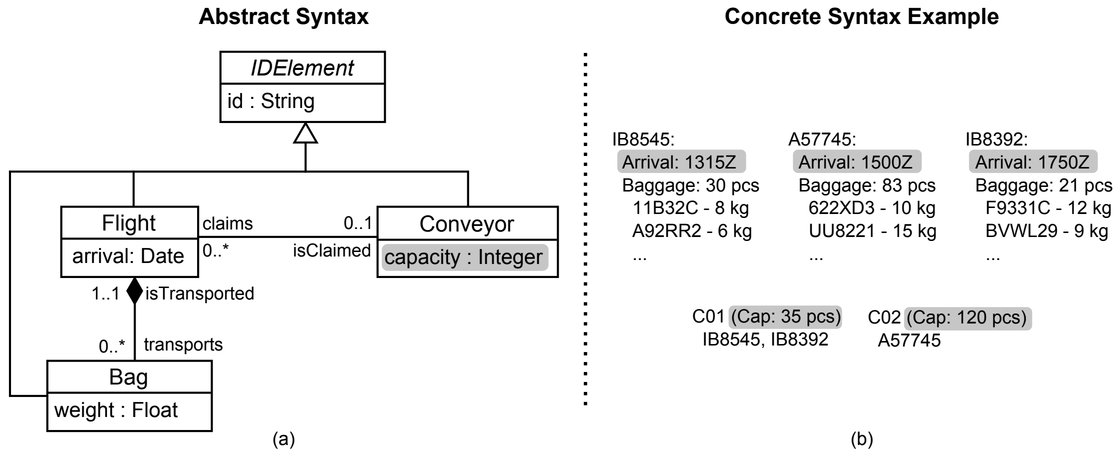


Figure 1. Abstract syntax and an example of concrete syntax of the *Baggage Claim* DSML (grey-filled boxes represent elements added after the collaboration).

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 not explicitly formalized.[3].

The development of a DSML usually consists in five different phases Mernik et al. (2005): 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, etc.) for the DSML. Clearly, the community around the language is a key element in the process. In this paper we use the term *community* to refer to what is known as *Communities of Practice*, which is defined as *groups of people informally bound together by shared expertise and passion for a joint enterprise* Tamburri et al. (2013). In this case, the DSML community covers the group of people involved in its 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.

As a running example, imagine the development of a DSML to facilitate the planification of the baggage claim service in airports. Let's assume that the airport baggage service needs to specify every morning the full daily assignment of flights to baggage claim conveyors so that operators can know well in advance how to configure the actual baggage system. For that, developers and domain experts (i.e., baggage managers) collaborate to define a DSML that serves this purpose.

Typically, domain experts are only involved at the very beginning and very end of the DSML development process. Assuming this is also the case for our example, during the analysis phase, developers would study the domain with the help of the baggage managers and decide that the DSML should include concepts such as `Flight`, `Bag` and `Conveyor`s to organize the baggage delivery. Developers would design and later implement the tooling of the language, thus coming up with a textual DSML like, for instance, the one shown in Figure 1 (both abstract and concrete syntax proposals are shown, except for the elements included in grey-filled boxes that are added later as explained in what follows).

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 might detect problems in the DSML 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 a construct to represent the capacity of conveyors, that may help them to perform a better assignment. Developers can also overlook design constraints and recommendations that could improve the DSML quality. For instance, constructs in the abstract syntax not having a concrete syntax definition could become an issue (e.g., `arrival` attribute in `Flight` concept).

The collaboration tasks can go beyond the definition of new DSML and can cover the usage of well-known GPLs, like UML. Let's imagine for instance the collaborative definition of class diagrams in

[3]The collaborative definition of the DSML semantics is out of the scope of this paper and has been considered as part of future work.

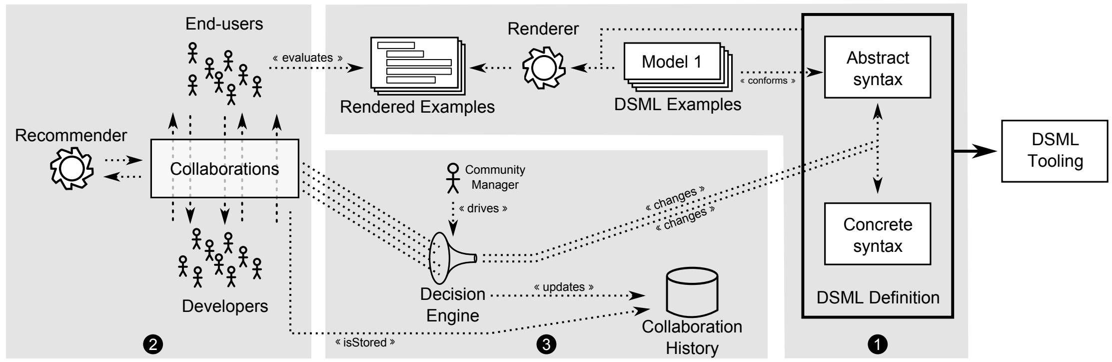


Figure 2. Collaborative development of DSMLs.

order to identify the domain of a new software artefact. In fact, we could even reuse the running example to ilustrate this scenario. Thus, the definition of the abstract syntax of the previous DSML requires the collaborative creation of a UML class diagram. In this sense, end-users (i.e., domain experts) use a common language (i.e., UML) to create a new model required for a particular purpose (in this case, the definition of a DSML). As before, end-users can propose changes to the model, which can after be discussed and eventually accepted/rejected in the final version.

Our aim is to incorporate the community collaboration aspect into all DSML definition phases, making the phases of the process more participative and promoting the early detection of possible bugs or problems. As we will see, this support also enables de collaborative creation of models conforming to modeling languages. Next section will present our approach.

MAKING DSML DEVELOPMENT COLLABORATIVE

We propose a collaborative approach to develop DSMLs following the process summarized in Figure 2. Roughly speaking, the process is as follows. Once there is an agreement to create the 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 are also defined by the developers to facilitate an example-based discussion, usually easier for non-technical users. 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). During this step, a recommender system (see *Recommender*) also checks the current DSML definition to spot possible issues according to quality metrics for DSMLs. If the recommender system detects possible improvements, it will create new proposals to be also discussed by the community. 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 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 agreement is reached, the contents of the solution are incorporated into the language, thus creating a new version.

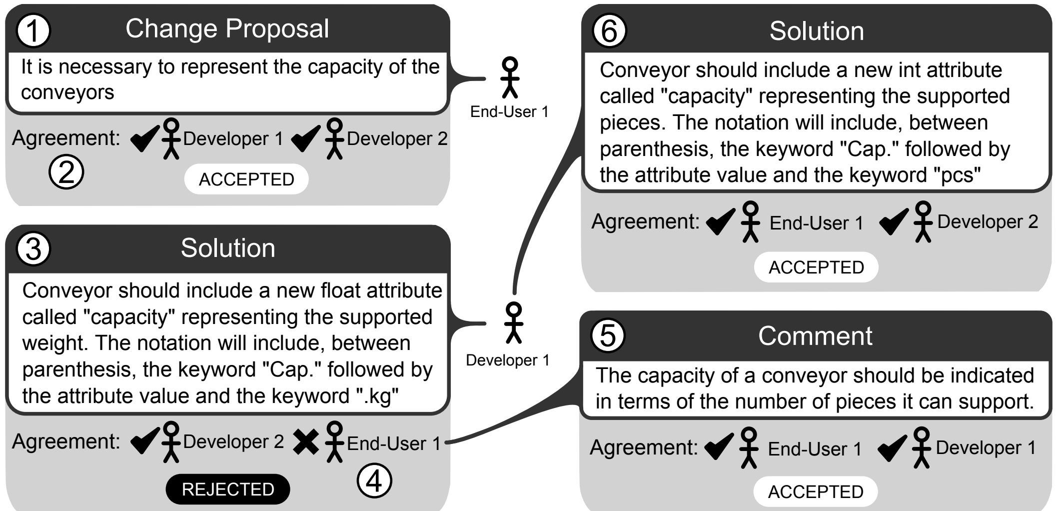


Figure 3. Example of collaboration in the *Baggage Claim* DSML.

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 comply with 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 when the language does not comply with commonly applied quality patterns, developers can be sure that it 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. Thus, at any moment, this traceability information can be queried (e.g., using standard OCL Object Management Group (OMG) (2015a) expressions) 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 *Baggage Claim* DSML mentioned above could have been the result of the imaginary collaboration scenario depicted in Figure 3. After developers completed a first version of the language, the collaboration begins with a community member detecting the need of expressing the capacity of the conveyors. 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 information. The member that identified the problem would create a change proposal with that aim, and 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 1. Clearly, it is important to make this collaboration iterations as quick as possible and with the participation of all the 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.

The recommender system may also participate in the collaboration and eventually improve the DSML. After checking the DSML definition, the recommender may detect that not all the attributes in the abstract syntax have a direct representation in the concrete syntax, as it happens with the `arrival` attribute of the `Flight` concept (as we will explain later, this is the result of applying the metric called *Symbol Deficit*). Thus, the system may create a new proposal informing about the situation and then the community could vote and eventually decide whether the DSML has to be modified.

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:

- Enabling the discussion about DSML elements,
- providing the metaclasses for representing collaborations and giving support to the decision-making process,
- providing a metric-based recommender that can help to develop high-quality DSMLs.

Representing the Elements of a DSML

To be able to discuss about changes on the DSML to-be, we must be able to represent both its abstract syntax (i.e., the concepts of the DSML) and its concrete syntax (the notation to represent those concepts) elements. Additionally, to improve the understanding of how changes in its definition affect the DSML, we provide a mechanism to automatically render DSML examples using the concrete syntax notation under development.

Abstract Syntax

The abstract syntax of a DSML is commonly defined by means of a metamodel written using a metamodeling language (e.g., MOF Object Management Group (OMG) (2015b) or Ecore Steinberg et al. (2008)). Metamodeling 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 metamodeling language, on which we rely to represent the abstract syntax of DSMLs.

Concrete Syntax

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 notational elements proposed for the language. Thanks to this, 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). The type of notational elements to represent largely depends on the kind of concrete syntax envisioned (textual or graphical). Nowadays, there are some tool-specific metamodels to represent graphical and textual concrete syntaxes (like the ones included in GMF [4] and Xtext [5]), or to interchange model-level diagrams Object Management Group (OMG) (2014b). However, a generic metamodel covering both graphical and textual syntaxes (and combination of both) is still missing. Therefore, we contribute in this paper 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 not 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 notation 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 (`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 `separator` of the `Value` metaclass allows defining the separator for multivalued elements. The `Composite` element can be used to define complex concrete syntax

[4] http://eclipse.org/gmf-tooling
[5] http://eclipse.org/Xtext

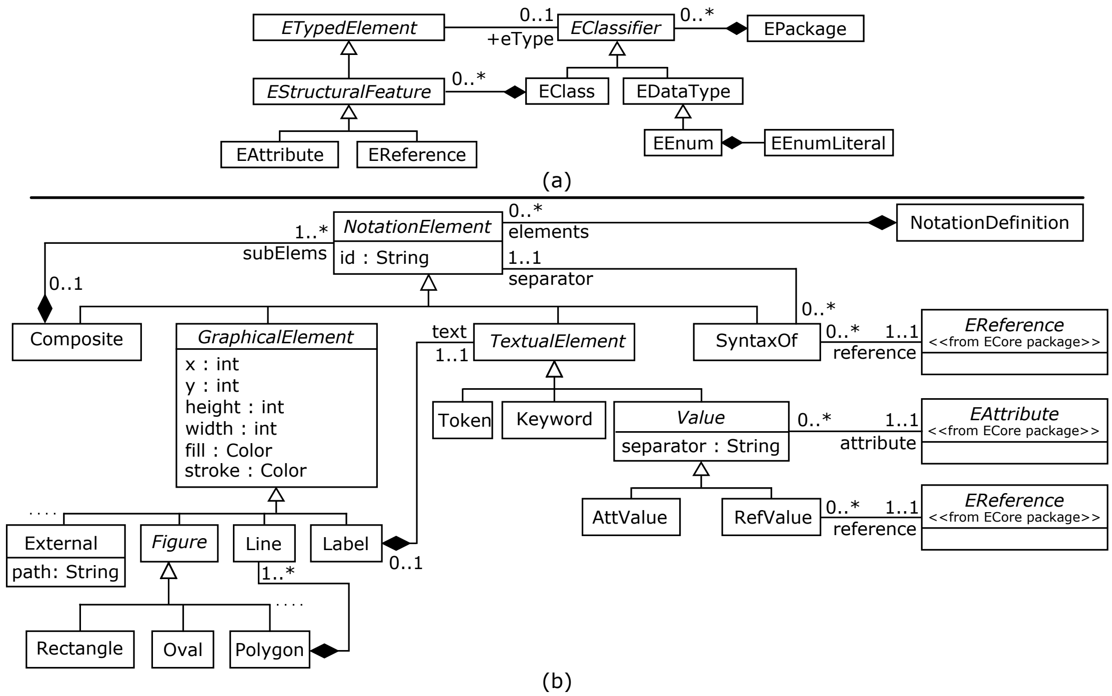


Figure 4. Excerpts of the (a) Ecore and (b) notation metamodels used to represent, respectively, the abstract and concrete syntaxes of DSMLs in Collaboro.

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

Renderer

The current DSML notation specification plus the set of example models for the DSML (expressed as instances of the DSML abstract syntax) can be used to generate concrete visual examples that help community members get a better idea of the language being built. We refer to this generator as *renderer*. The renderer takes, as inputs: (1) the abstract and (2) concrete syntaxes of the DSML, and (3) the set of example models conforming to the abstract syntax; and returns a set of images representing the example models expressed according to the concrete syntax defined in the notation model (additional technical details about the render process will be given in Section).

We believe the advantages of this approach is twofold. On the one hand, it is a lightweight mechanism to quickly validate the DSML without generating the DSML tooling support. On the other hand, developers and end-users participating in the collaboration can easily assess how the language looks like without the burden of dealing with the abstract and concrete syntax of DSML, which are expressed as metamodels.

Example

Figure 1a shows an example of the abstract syntax for the *Baggage Claim* DSML while Figure 5 shows the notation model for the textual representation of the metaclass `Conveyor` of such DSML (Figure 5a shows a textual example and Figure 5b shows the corresponding notation model). Note that `AttValue` and `RefValue` metaclass instances are referring to elements from the abstract syntax metamodel. Figure 1b shows a possible renderization of a model for such language.

Representing the Collaborations

The third metamodel required in our process focuses on representing the collaborations that annotate/modify the DSML elements described before. This collaboration metamodel, which is shown in Figure 6, allows representing both static (e.g., change proposals) and dynamic (e.g., voting) aspects of the

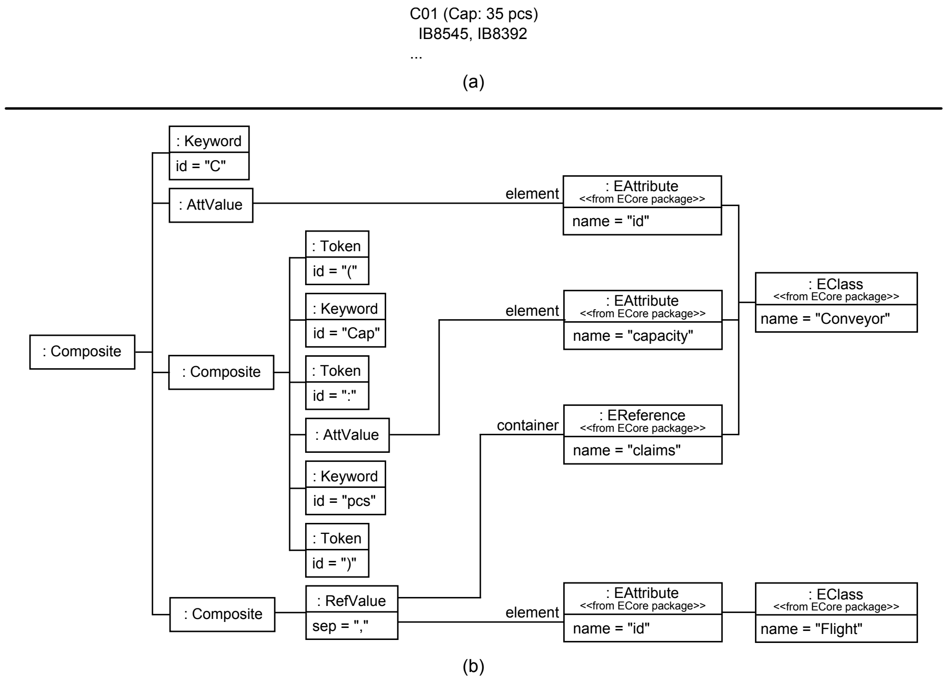


Figure 5. (a) Textual representation example of the metaclass `Conveyor` of the *Baggage Claim* DSML and (b) the corresponding notation model.

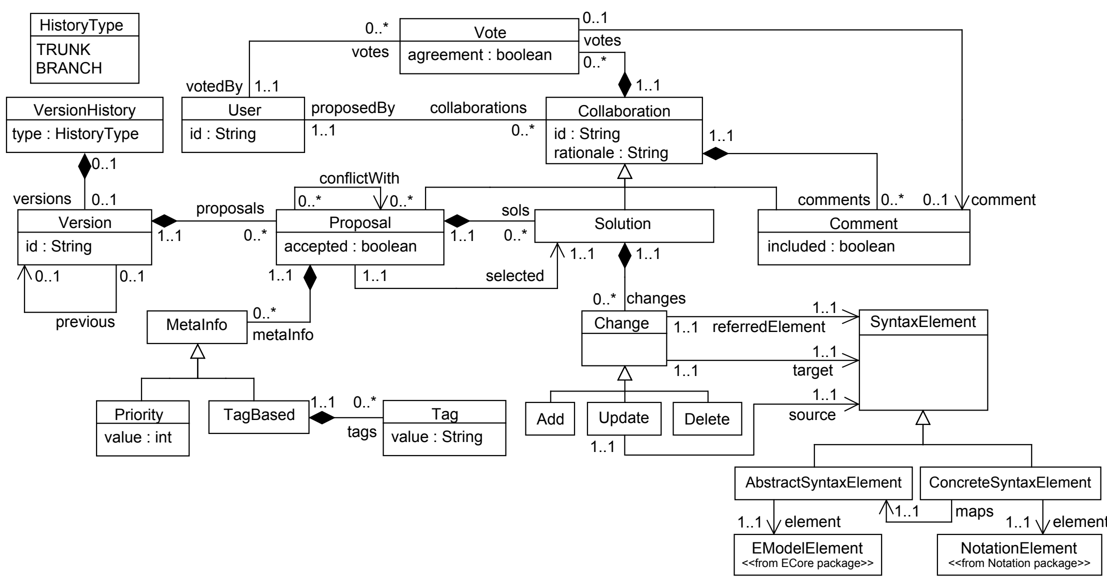


Figure 6. Core elements of the Collaboro metamodel.

collaboration. Being the core of our collaborative approach, we refer to this metamodel as the *Collaboro metamodel*.

Static Aspects

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 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` metaclasses have a reference linking to the element which is being changed (`element` reference). Changes in the abstract syntax are expressed in terms of the metamodeling language (i.e., `EModelElement` 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 `Change` metaclass 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` reference 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` reference refers to the element which contains the element to be updated (e.g., a metaclass) whereas `target` reference 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 Aspects

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.

Making Decisions

Community votes are used to decide which collaborations are accepted and must be incorporated into the language. Collaboration models include all the necessary 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.

Example

As example of collaboration, we show in Figure 7 the collaboration model which would be obtained when using Collaboro to model the example discussed previously. The figure is divided in several parts according to the collaboration steps enumerated previously. For the sake of clarity, references to `User` metaclass instances have been represented as string-based attributes and the `rationale` attribute is

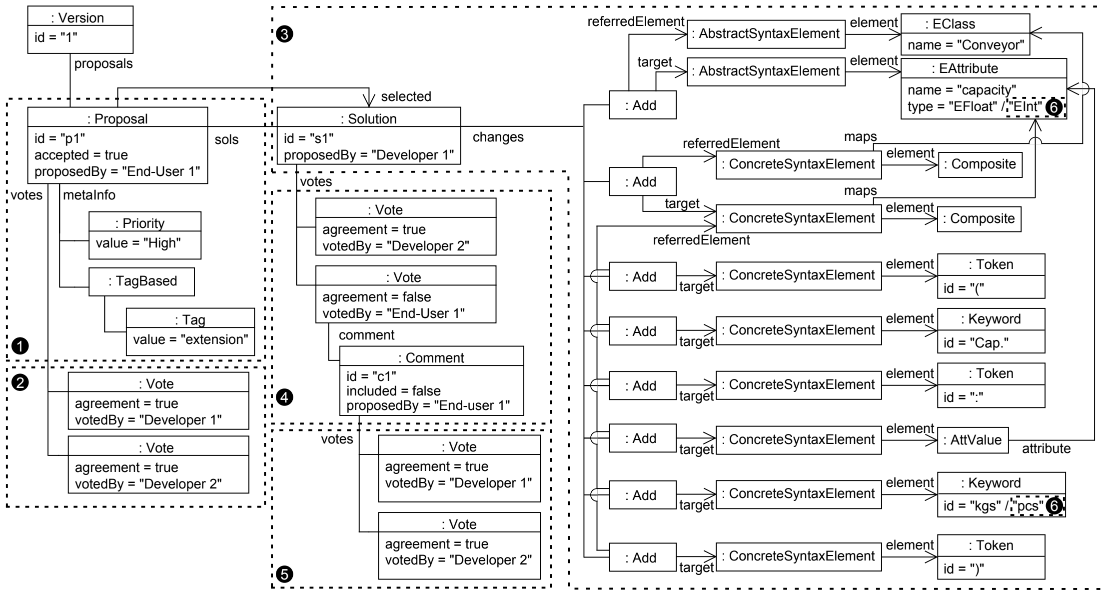


Figure 7. The collaboration model representing the collaborations arisen in the *Baggage Claim* DSML.

not shown. The figure shows the collaboration as an instance of the Collaboro metamodel. At the tool level, we offer a user-friendly interface enabling all kinds of users to easily contribute and vote during the discussion process while the tool updates the collaboration model behind the scenes in response to the user events.

Part 1 of Figure 7 shows the collaboration model just after *End-User 1* makes the request. It includes a new `Proposal` instance that is voted positively by the rest of the users and therefore accepted (see part 2). Then, a new solution is proposed by *Developer 1* (see part 3), which involves enriching the `Conveyor` metaclass with a float attribute in addition to define the concrete syntax. However, this solution is not accepted by all the community members: *End-User 1* does not agree and explains his disagreement (see part 4). Since the comment is accepted (see part 5), *Developer 1* updates the solution to incorporate the community recommendations (see part 6). Note that the elements describing the model changes in parts 3 and 6 are mutually exclusive. Moreover, the `included` attribute of the `Comment` element in part 4 will be activated as a consequence of the solution update. Once everybody agrees on the improved solution, it is selected as the final one for the proposal (see the `selected` reference).

Now the development team can modify the DSML tooling knowing that the community needs such change and agrees on how it 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 `Conveyor` metaclass was changed.

Metric-based Recommender

When developing DSMLs, several quality issues regarding the abstract and concrete syntaxes can be overlooked during the collaboration. While developers are maybe the main responsibles for checking that the language is being developed properly, it is important to note that these issues may arise from both developers (e.g., they can forget defining how some concepts are represented in the notation) and end-users (e.g., they may miss that the notation is becoming too complicated for them to later being able to manage complex models). We propose to help both developers and end-users to develop better DSMLs by means of a recommender engine which checks the language under development to spot possible issues and improvements.

The recommender applies a set of metrics on the DSML to check its quality, in particular, to ensure that the resulting language is expressive, effective and accurate enough to be well-understood by the end-users. Metrics can target both the abstract and concrete syntaxes of a DSML. Concrete syntax metrics can in turn target either textual or graphical syntaxes. While several metrics for abstract and textual concrete syntaxes have been devised in previous works Cho and Gray (2011); Aguilera et al. (2012); Power and Malloy (2004); Crepinšek et al. (2010), the definition and implementation of metrics for graphical concrete syntaxes is still an emerging working area. Thus, in this work, we explore how metrics for abstract and concrete syntaxes can be implemented in our approach, but we mainly focus on those ones regarding graphical concrete syntaxes.

Abstract Syntax Metrics

The abstract syntax of a DSML is defined by a metamodel, as commented before. While the identification of proper DSML constructs (i.e., concepts and relationships) usually relies on the domain experts, identifying and solving design issues (e.g., creating hierarchies to promote extensibility or identifying patterns such as factoring attributes) is normally performed by the developers. Thus, to provide consistent solutions for recurring metamodel design issues, some metrics applied to abstract syntax metamodels may offer key insights on its quality.

There are currently several works providing a set of metrics for metamodels as well as for UML class diagrams that can be applied in this context (e.g., Cho and Gray (2011); Aguilera et al. (2012)). As a proof of concept to evaluate the abstract syntax of DSMLs in our approach, we implemented a couple of metrics that validate hierarchical structures in metamodels (inspired by Aguilera et al. (2012)). Thus, we consider that such structures are invalid whether either there is only one derived class or whether an inheritage is redundant (i.e., already covered by a chain of inheritage). As our approach relies on Ecore, other metrics defined for this metamodeling language could be easily plugged in by using the extension mechanism provided, as we will show afterwards.

Concrete Syntax Metrics

The concrete syntax of a DSML can be textual or graphical (or hybrid). As textual DSMLs are usually defined by means of a grammar-based approach, which is also the case for General-Purpose Languages (GPLs), existing support for evaluating the quality of GPLs could be applied (e.g., Power and Malloy (2004) and Crepinšek et al. (2010)). Apart from this GPL-related support, the current support to assess the quality of the concrete syntaxes in the DSML field is pretty limited. Thus, in this paper, we apply a unifying approach to check the quality of any DSML concrete syntax (i.e., textual and/or graphical).

With this purpose, we employ the set of metrics based on the cognitive dimensions framework Green (1989), later formalized in Moody (2009), where metrics are presented according to nine principles, namely: cognitive integration, cognitive fit, manageable complexity, perceptual discriminality, semiotic clarity, dual coding, graphic economy, visual expressiveness and semantic transparency. Several works have applied them to specific DSMLs (e.g., Genon et al. (2011b) or Le Pallec and Dupuy-Chessa (2013)). Nevertheless, none of them has tried to implement such metrics in a way that can be applied generically to any DSML. As Collaboro provides the required infrastructure to represent concrete syntax at a technology-agnostic level, we propose to define a set of DSML metrics adapted from Moody's principles for designing cognitively effective notations. In the following, we present how we addressed five of the nine principles to be applied to our metamodels, and we justify why the rest of the principles were discarded.

Semiotic clarity. This principle refers to the need of having a one-to-one correspondence between notation symbols and their corresponding concepts, thus maximizing precision and promoting expressiveness. We can identify four metrics according to the possible situations that could appear: (1) *symbol deficit*, when a concept is not represented by a notation symbol (sometimes this situation could be evaluated positively as to avoid having too many symbols and losing visual expressiveness); (2) *symbol excess*, when a notation symbol does not represent any concept; (3) *symbol redundancy*, when multiple notation symbols can be used to represent a single concept; and (4) *symbol overload*, when multiple concepts are represented by the same notation symbol.

In Collaboro, these metrics can be computed by analyzing the mapping between the abstract syntax elements and the notation model elements of the DSML. On the one hand, the analysis of the abstract syntax consists on a kind of flattening process where all the concepts are enriched to include the attributes and references inherited from their ancestors. The aim is to identify the DSMLs elements (i.e., concept, attribute or reference) for which a concrete syntax element has to be defined. On the other hand, the analysis of the concrete syntax focuses on the discovery of symbols. When a symbol uses multiples graphical elements to be represented (e.g., using nested `Composite` elements or `SyntaxOf` elements), they are aggregated. The result of this analysis is stored in a map that links every abstract syntax element with the corresponding concrete syntax element, thus facilitating the calculation of the previous metrics. This map will be also used in the computation of the remainder metrics.

Visual Expressiveness. This principle refers to the number of visual variables used in the notation of a DSML. Visual variables define the dimensions that can be used to create notation symbols (e.g., shape, size, color, etc.). Thus, to promote its visual expressivenes, a language should use the full range and capacities of visual variables.

To assess this principle, we define a metric which analyzes how visual variables are used in a DSML. The metric leverages on the previous map data structure and enriches it to include the main visual variables used in each symbol. According to the current support for visual variables of the notation metamodel (recall `GraphicalElement` metaclass attributes), these variables include: size (`height` and `width` attributes), color (`fill` and `stroke` attributes) and shape (subclasses of `GraphicalElement` metaclass). The metric checks the range of visual variables used in the symbols of the DSML and notifies the community when the notation should use more visual variables and/or more values of a specific visual variable to cover the full range.

Graphic Economy. This principle states that the number of notation symbols should be cognitively manageable. Note that there is not an objective rule to measure the complexity of notation elements (e.g., expert users may cognitively manage more symbols than a novice). There is the *six symbol* rule Miller (1956) which states that there should be no more than six notation symbols if only a single visual variable is used. We therefore devised a metric based on this rule to assess the level of graphic economy in a DSML.

Perceptual Discriminality. This principle states that different symbols should be clearly distinguishable from each other. Discriminality is primarily determined by the visual distance between symbols, that is, the number of visual variables on which they differ and the size of these differences. This principle also states that every symbol should have at least one unique value for each visual variable used (e.g., unique colors for each symbol). Thus, to assess the perceptual discriminality, we define a metric which also relies on the previous map data structure, compares each pair of symbols and calculates the visual distance between them according to the supported visual variables (i.e., number of different visual variables per pair of symbols). By default, the metric notifies the community when the average distance is lower than one, but it can be parameterized.

Dual Coding. This principle suggests that using text and graphics together conveys the information in a more effective way. Textual encoding should be then used in addition of graphical encoding to improve understanding. However, textual encoding should not be the only way to distinguish between symbols. We defined a metric that checks whether each symbol uses text and graphics elements, thus promoting dual coding. To this aim, we leverage on our notation metamodel, which allows to attach textual elements to symbols by employing `Label` elements that contain `TextualElement` elements. This metric notifies the community when more than a half of the symbols are not using both text and graphics.

The remaining four Moody's principles were not addressed due to the reasons described below.

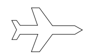
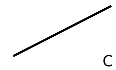
Semiotic Transparency. This principle states that a notation symbol should suggest its meaning. This principle is difficult to evaluate as it relies on many parameters such as context and good practices in the specific domain. Furthermore, as the meaning of a representation is subjective, an automatic verification of this principle would be difficult to reach.

Complexity Management. This principle refers to the ability of the notation to represent information without overloading the human mind (e.g., providing hierarchical notations). Although this could be addressed in the notation model by providing mechanisms for modularization and hierarchical structuring, we believe that assessing this principle strongly depends on the profile and background of the DSML end-users and it is therefore hard to measure.

Cognitive Integration. This principle states that the visual notation should include explicit mechanisms to support integration of information from different diagrams. In this sense, this principle refers to the results of composing different DSMLs, which is not an scenario targeted by our approach.

Cognitive Fit. This principle promotes the fact that different representations of information are suitable for different tasks and audiences (e.g., providing different concrete syntaxes for the same abstract syntax). Like in the complexity management principle, assessing the cognitive fit of the notations of a DSML is

Table 1. Example of Visual Expressiveness and Perceptual Discriminality for the *Baggage Claim* DSML. V E : Visual Expressiveness, P D : Perceptual Discriminability

		 B	 C	V E
Shape	Polygon	Rectangle	Line	3/5
Size	H : 5 W : 9	H : 5 W : 9	H : 1 W : 12	2
Color	Fill : White Stroke : Black	Fill : Black Stroke : Black	Fill : White Stroke : Black	2/49
P D	Visual Distance B : 2 C : 2	Visual Distance A : 2 C : 3	Visual Distance A : 2 B : 3	

directly related to the expertise of the different communities using the language, which is hard to measure with an automatic evaluation.

Recommending Changes

The results of the previously shown metrics provide the community developing a DSML with an important feedback to address potential improvements. In Collaboro, the DSML development process incorporates a recommender that plays the role of a user in the collaboration process. This "recommender user" can check the different versions of the DSML under development according to the previously shown metrics and propose new changes identifying the weak points to be discussed in the community. Metrics can be deactivated if wished and can be given different relevance values that can also be used to sum up the results to calculate a general value assessing the quality of the DSML under development.

Example

In this section, we will show an example of the metrics regarding visual expressiveness and perceptual discriminatity for the *Baggage Claim* DSML. For the sake of illustration purposes, we describe these metrics on an alternative graphical syntax to the DSML, where the `Flight` concept is represented as a poligon with the shape of an airplane, the `Conveyor` concept is represented as a black filled-rectangle and the `claims` reference is represented as a line. The computation of these metrics are specially tailored to the visual variables supported by our notation metamodel. Table 1 illustrates how these two metrics are calculated. As can be seen, visual expressiveness results assess the number of different values used for each visual variable. Thus, there are three out of five values for the shape dimension, two different values for the size dimension and two different values for the color dimension. On the other hand, the visual distance is calculated for each pair of symbols and measures the number of different visual variables between them. For instance, the black-filled rectangle differs in two visual variables (i.e., color and shape) with the airplane polygon; and all the supported visual variables with regard to the line. These results reveal a good visual expressiveness (good values for shape and size visual variables while the color range is appropriate for the number of symbols) and perceptual discriminality (visual distance is in average more than 2, where the highest value is 3) therefore validating this graphical notation proposal.

COLLABORATIVE MODELING

In this section we will show how our approach could be easily adapted to support collaborative modeling. This adaptation is depicted in Figure 8. Unlike the Figure 2, where we illustrated the process for the collaborative development of DSMLs, in this case the community evaluates and discuss changes about the model being developed and not the metamodel. Thus, once there is a first version of the model and a set of examples (step 1), the community discusses how to improve the models (step 2). The discussion arises changes and improvements, that have to be voted and eventually incorporated in the model (step 3). Discussion and decisions are recorded (see *Collaboration History*), thus keeping track of the modifications performed in the model.

To support this development process, the modifications to perform in the original Collaboro metamodel are very small. Figure 9 shows the new metamodel to track the collaboration. As can be seen, the only

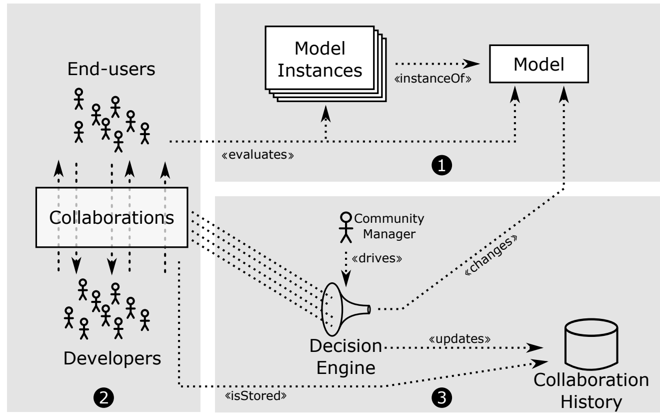


Figure 8. Collaborative modeling.

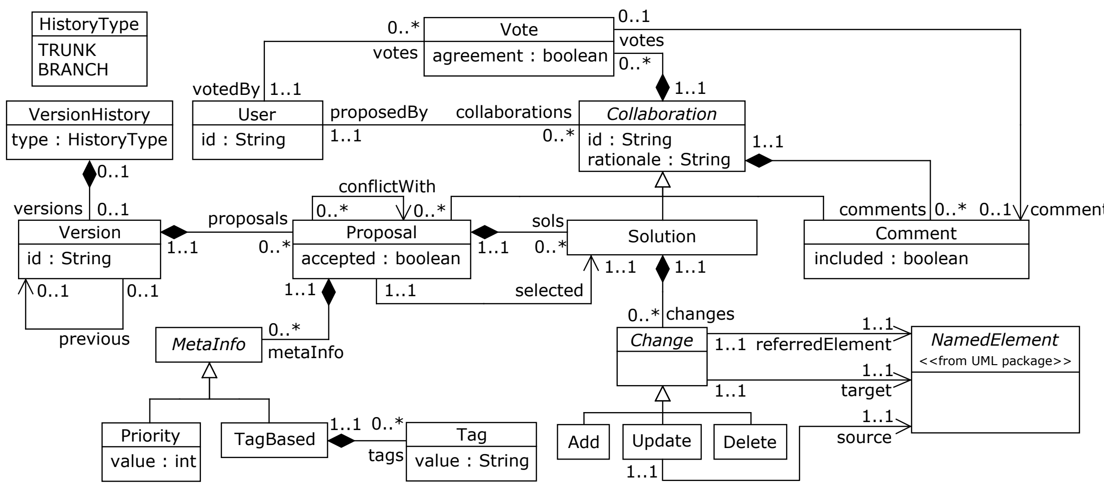


Figure 9. Core elements of the adaptedCollaboro metamodel.

changed element is the *SyntaxElement*, which now has to refer to the main (i.e. root) metaclass of the modeling language being used to link the model elements with their metamodel definition. For instance, by default, the Figure includes the element *NamedElement* from UML, thus illustrating how Collaboro could be used for the collaborative development of UML models. Other languages could be supported following this same approach.

TOOL SUPPORT

Since the very first implementation of Collaboro was released, the tool support has evolved to integrate the full set of features described in this paper[6]. The new architecture of the developed tool is illustrated in Figure 10. The main functionalities of our approach are implemented by the backend (see *Collaboro Backend*), which includes specific components for modeling both the DSML elements and the collaborations (see *Modeling Support*), rendering the notation examples (see *Notation Renderer*) making decisions (see *Decision Engine*), and recommending changes (see *Recommender System*). As front-end for Collaboro, we have developed two alternatives: (1) a web-based front-end, which gives access to the collaboration infrastructure from any web browser; and (2) an Eclipse-based front-end, which extends the platform with views and editors facilitating the collaboration. Next, we describe in detail each component of this architecture.

[6]The tool is available at http://som-research.github.io/collaboro

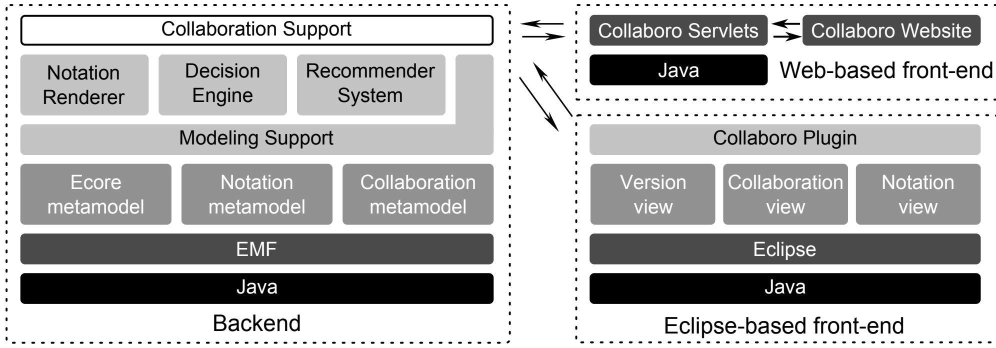


Figure 10. Architecture of Collaboro tool support.

Collaboro Backend

This component provides the basic functionality to develop collaborative DSMLs as explained in this paper. Collaboro relies on the EMF framework Steinberg et al. (2008) (the standard *de facto* modeling framework nowadays) to manage the models required during the development process. In the following, we describe the main elements of this component.

Modeling Support

Collaboro provides support for managing models representing the abstract and concrete syntaxes, and the collaboration models. We implemented the metamodels described in previous sections as Ecore models (the metamodeling language used in EMF) and provided the required API. To support concurrent collaboration the tool can be configured to store the models in a CDO[7] model repository.

Notation Renderer

The tool incorporates a generator which automatically creates the graphical/textual representation of the DSML example models. This component enables the lightweight creation of SVG SVG (2011) images from notation models to help users "see" how the notation they are discussing will look like when used to define models with that DSML. The generator analyzes each example model element, locates its abstract/concrete syntax elements and interprets the concrete syntax definition to render its textual/graphical representation. `GraphicalElement` and `TextualElement` concrete syntax elements indicate the graphical or textual representation to be applied (e.g., a figure or a text field), while `Composite` and `SyntaxOf` concrete syntax elements are used for layout and composite elements.

Decision Engine

This component is responsible for updating the dynamic part of the collaboration models (recall the support for votes and decisions). The current support of the tool implements a total agreement strategy to infer community agreements from the voting information of the collaboration models.

Recommender System

This component provides the required infrastructure to calculate both abstract and concrete syntaxes metrics in order to ensure their quality. The recommender is executed on demand by the community manager. The current support of the tool implements metrics to evaluate the quality of concrete graphical syntax issues.

New metrics can be plugged in by extending the Java elements presented in Figure 11. The entry point is the `Metric Factory` class, which is created for each DSML and is responsible for providing the list of available metrics. `Metric`s have a name, a description, a dimension (e.g., each Moody's principle), an activation, a priority level and an acceptance ratio. The acceptance ratio allows specifying the maximum number of elements of syntaxes that can be wrong (e.g., not conforming to the metric). Every metric also includes an `execute()` method for the recommender to compute them. This function returns a list of `MetricResult`s describing the assessment of the metric. Metric results includes a status (i.e., measured in three levels), a reason describing the assessment in natural language and a ratio of fulfillment for the metric. Metric results also include a list of `ReferredElement`s pointing to those abstract or

[7] http://www.eclipse.org/cdo

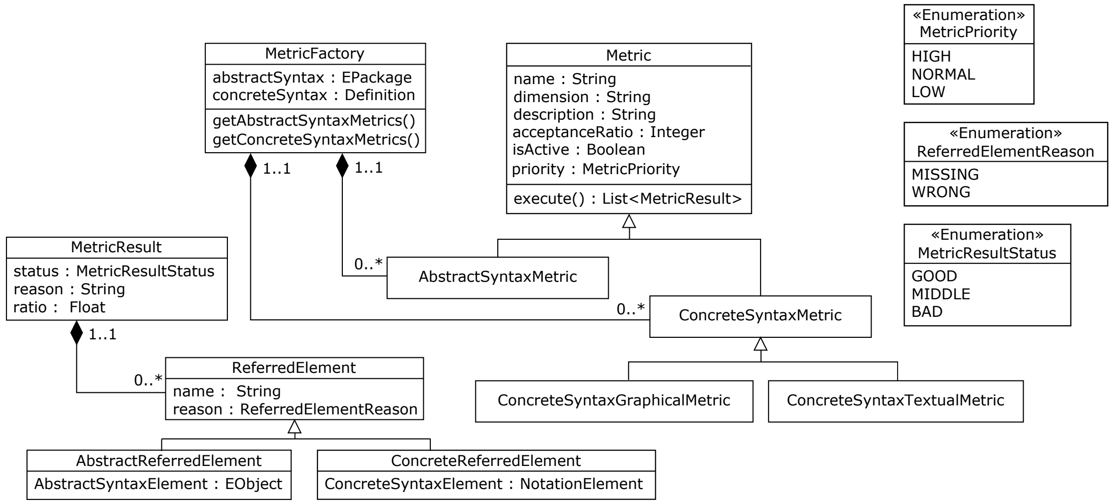


Figure 11. Core elements of the recommender engine.

concrete syntaxes elements not conforming with the metrics being calculated, thus helping developers to spot the DSML elements not satisfying each metric (if any).

Eclipse plugin

We have developed an Eclipse plugin implementing the Collaboro process and DSML. 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 12a includes a snapshot of the environment showing the last step of the collaboration described in Section . 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 12a. Finally, the *Notation view* uses the notation generator to render a full example model of the language. For instance, the Notation view in Figure 12b shows the notation for an example model, which allowed detecting the missing attribute regarding the conveyor capacity.

Web-based front-end

The developed web support includes two components: (1) the server-side part, which offers a set of services to access to the main functionalities of Collaboro; and the client-side part, which allows both end-users and developers to take part of the DSML development process from their browsers. The server-side component has been developed as a Java web application which uses a set of Servlets providing the required services. On the other hand, the client-side component has been developed as an AngularJS-enabled website.

Figure 13 shows a snapshot of the developed website. As can be seen in Figure 13a, the website follows an arrangement similar to that one used in the Eclipse plugin. Thus, on top, there are two sections showing the current status of (1) the abstract syntax of the DSML on the left and (2) several model examples rendered with the concrete syntax definition of the DSML on the right (both sections are zoom-enabled). These sections include several pictures that can be navigated by the user (e.g., it is possible to evaluate the different example models rendered). At the bottom of the website, there are two more sections aimed at managing the collaborations, in particular, (1) a tree including all the collaboration elements on the left and (2) a details view on the right which shows the information of a collaboration once it is selected in the tree. Furthermore, the tree view also includes buttons to create, edit and delete collaborations.

The website also includes a left menu bar which allows the user to navigate through the different versions of the DSML as well as indicate some information about the recommender system status. Additionally, the user can quickly see the number of issues detected by the recommender, configure the metrics (see Figure 13b) that have to be executed and perform the metric execution to incorporate the change proposals into the collaboration.

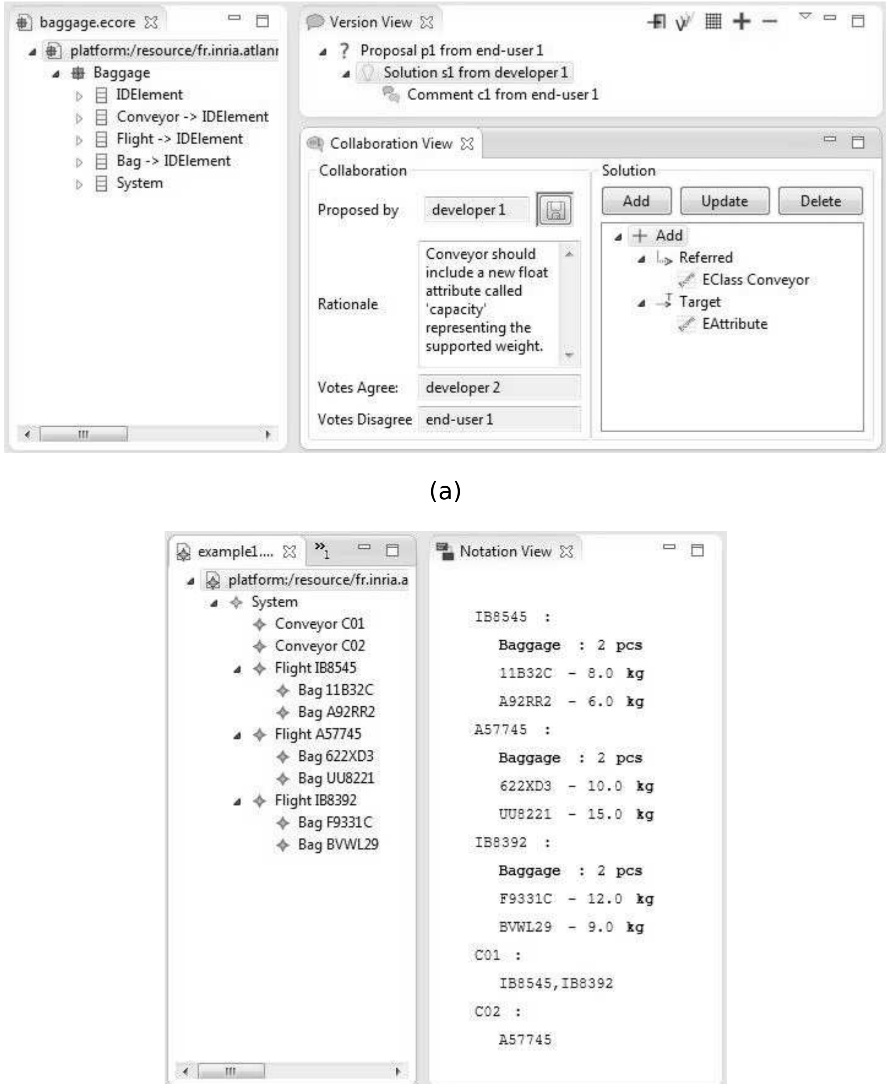


Figure 12. (a) Snapshot of the Collaboro Eclipse plugin. (b) Collaboro Eclipse plugin with the *Notation view* rendering the concrete syntax for a model.

APPLICATION SCENARIOS

In this section, we report the use of Collaboro in two types of scenarios: (1) the creation of new DSMLs, based on two different case studies; and (2) the extension of existing DSMLs, where we describe our experience in one case study. We also mention some lessons learned in the process.

Developing new DSMLs

We used Collaboro in the creation of two new DSMLs: (1) a textual DSML to define workflows and (2) three metamodels to represent code hosting platforms in the context of a modernization process. We explain each case in the following.

Creating a Textual DSML

Collaboro was used in the development of a new DSML for MoDisco[8], an Eclipse project aimed at defining a group of tools for Model-Driven Reverse Engineering (MDRE) processes. The goal of this new DSML is to facilitate the development of MDRE workflows that chain several atomic reverse engineering

[8]http://eclipse.org/modisco

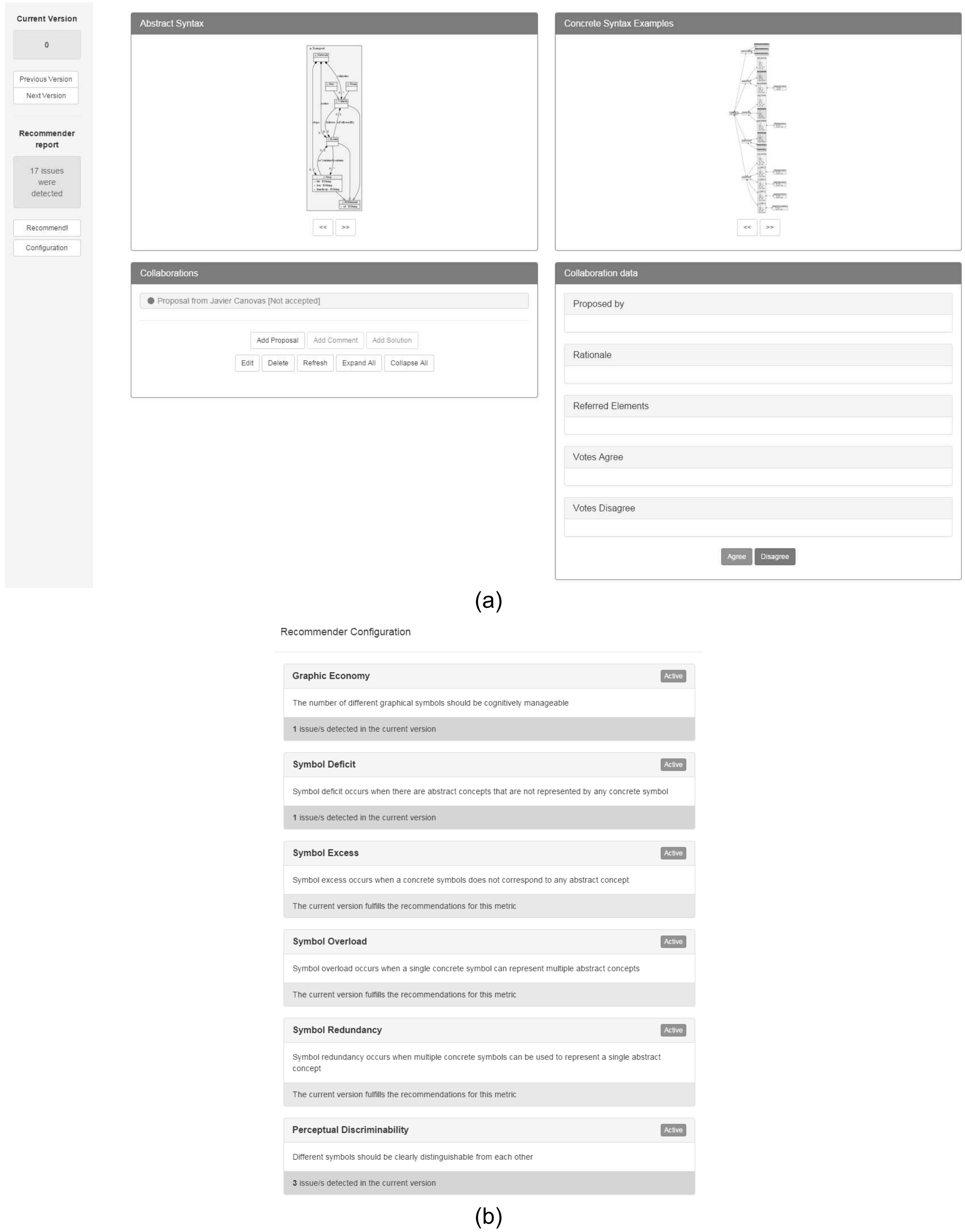


Figure 13. Snapshots of the (a) Collaboro web client and (b) a subset of supported metrics.

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. 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 the role of end-users in the process. One of the members was in a different country during the collaboration so only asynchronous communication was possible.

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). The collaboration regarding the abstract syntax involved changes in concepts and reference cardinalities, while regarding the concrete syntax, the community chose to go for a textual-based notation and mainly discussed around the best keywords or style to be used for that.

Defining metamodels

We have also applied Collaboro for defining a set of metamodels used in a model-driven re-engineering process (i.e., only the abstract syntax of the DSML was part of the experiment since the models were to be automatically created during the reverse engineering process). In particular, the process was intented to provide support for migrating Google Code to GitHub projects, thus requiring the corresponding PSM metamodels for both platforms, plus a PIM metamodel to represent such projects at high level of abstraction (following the typical terminology defined by the Model-Driven Architecture (MDA) approach from the OMG Object Management Group (OMG) (2014a)). As the developers were distributed across different geographical locations, we decided to use Collaboro to create the PSM and PIM metamodels required.

Six researchers geographically dispersed (i.e., the participants were part of three research groups, making three groups composed of 3, 2 and 1 researchers) collaborated in the definition of the metamodels. To kickstart the collaboration, one of the teams created a first version of each metamodel. As the collaboration was focused on defining only the abstract syntax of the language, there was no need for creating a notation model, and therefore the set of examples were rendered following a class-like diagram style. The collaboration took three weeks and resulted in two versions for each one of the PSM metamodels and only one version for the PIM metamodel since there the agreement was faster.

Extending an Existing DSML

More recently, we were contacted by a community member of the Architecture Analysis & Design Language (AADL) [9], and one of the lead developers in charge of defining an extension to such language. AADL is an architecture description language used in the embedded and real-time systems field. It is a textual DSML with large abstract and concrete syntaxes. The abstract syntax contains more than 270 concepts and the concrete syntax is composed of more that 153 elements (including keywords and tokens). The language was being extended to incorporate support for behavior specification. This extension, called AADL Behavior Annex (AADL-BA) [10], was being defined as a plugin enriching both the abstract and concrete syntaxes.

At the time, the definition of the extension was taken care by a standarization committee open to new contributions. Change proposals were informally managed by in-person voting (i.e., raising hands in a meeting) or online ballots. Later, the documentation of the change proposal was spread out in a document, presentation or online wiki documentation. As explained to us by this lead developer, this process made tracking modifications very hard in the language as well as the corresponding argumentations, and he proposed to use Collaboro to manage the development of the extension for AADL. As a first step, we created a fake AADL project so that this person could play around with the tool and assess its usefulness for the AADL community. The feedback was that the tool would be very useful for the project at hand if we were able to deal with some technical challenges linked to the current setting used by the project so far. In particular, to be able to use Collaboro for managing the ADDL-BA language definition process we needed to import: (1) previous discussions stored in the wiki-based platform and (2) the current concrete syntaxes of the AADL and AADL-BA language defined in Xtext and ANTLR respectively (the abstract syntax was already defined as an EMF model so it could be directly imported into Collaboro). It was also clear that to simplify the use of the tool, we had to provide a web interface since it would be too complex for the members of the AADL community to install an Eclipse environment just for the purpose of discussing around language issues.

In the end, time constraints prevented us to test the tool with AADL community at large (the AADL-BA committee meets at fixed dates and we did create a web-based interface but could not get a new version

[9] http://www.aadl.info

[10] http://penelope.enst.fr/aadl/wiki/Projects#AADL-BA-FrontEn

of the tool with all the scripts required to import the legacy data on time), but the private iterations with the AADL-BA developer and his validation and positive feedback helped us a lot to improve Collaboro and learn more about the challenges of using Collaboro as a part of an ongoing language development effort. We are still in contact with this community and we will see if we can complete the test in the future or reach out other similar standardization committees.

Lessons Learned

The development of the previous case studies provided us with some useful insights on the Collaboro process that since then have been integrated in our approach. For instance, in the first and second case studies, it turned out that conflicting proposals were frequent and therefore 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 fact together with the discussions on what should constitute a new version and when to end the discussions (e.g., what if there was unanimity but not everybody had voted, should we wait for that person? for how long?) helped us to realize the importance of an explicit community manager role in charge of making sure the collaboration is always fluid and there are no bottlenecks or deadlocks.

During the development of the three case studies, concurrent access to the models turned out to be a must as well since most of the time collaborations overlapped at some point. The experience gathered during the development of the first case study, where the collaboration was performed only in the Eclipse-based plugin, and later the requirements of the second and third case studies allowed us to provide a second front-end for the approach based on a web-client. Thus, the web-enabled support was crucial to allow all the developers to contribute and visualize how the metamodels evolved during the collaboration.

In all the case studies the notation view allowed the participants to quickly validate the concrete syntax. This is specially important since for non-technical users it is easier to discuss at the concrete syntax level than at the abstract level.

The only common complaint we got was regarding the limited support for voting (mainly raised in the first case study but also raised in the others), where participants reported that 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. We plan to incorporate extra support to define how to make decisions, in a similar way as proposed in Cánovas Izquierdo and Cabot (2015).

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* of a software product was introduced in Hess et al. (2008) and other authors have studied this collaboration as part of the requirement elicitation Mylopoulos et al. (1999), ontology development Leenheer (2009); Siorpaes (2007) and modeling phases of the software Hildenbrand et al. (2008); Lanubile et al. (2010); Whitehead (2007); Rittgen (2008), but neither of them focuses on the DSML language design process nor they present the collaboration as a process of discussion, voting and argumentation from the beginning to the end of the language development process. End-user participation is also the core of user-centered design Norman and Draper (1986), initially focused on the design of user interfaces but lately applied to other domains (e.g., agile methodologies Hussain et al. (2009) or web development Troyer and Leune (1998)). 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, some works propose to derive a first DSML definition by means of user demonstrations Cho et al. (2012); Kuhrmann (2011); Sánchez Cuadrado et al. (2012); López-Fernández et al. (2013) or grammar inference techniques Javed et al. (2008); Liu et al. (2012), where example models 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 to trigger the refinement process based on the discussions among the different users Cánovas Izquierdo et al. (2013).

Subsets of our proposal can also be linked to: i) specific tools for model versioning (e.g., AMOR repository[11] and Altmanninger et al. (2009)) that have already proposed a taxonomy of metamodel changes, ii) online-collaboration (Brosch et al. (2009); Gallardo et al. (2011)) promoting synchronous collaboration among developers, iii) metamodel-centric language definition approaches (Scheidgen (2008); Prinz et al. (2007)) where the concrete syntax is considered at the same level as the abstract one and iv) collaboration protocols Gallardo et al. (2012). 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). The agreed DSML definition at the end of the Collaboro process could be then the input of the complete DSML modeling environment aimed by some of the tools mentioned above.

Regarding the recommender engine and the calculation of metrics for DSMLs, we can identify works centered on assessing the quality of both the abstract and concrete syntaxes, and the main features of the language (e.g., reusability, integrability or compatibility). There are several works providing metrics to check the quality in metamodels Cho and Gray (2011); Aguilera et al. (2012) and in the notation used for textual DSMLs Power and Malloy (2004); Crepinšek et al. (2010). With regard to graphical DSMLs, Moody's principles Moody (2009) have emerged as the predominant theoretical paradigm. Originally based on the cognitive dimensions framework Blackwell et al. (2001); Green (1989); Green and Petre (1996), Moody's principles address their theoretical and practical limitations. While these principles provide a framework to evaluate visual notations, other works have put them into practice by analyzing DSMLs Genon et al. (2011b,a); Moody and Hillegersberg (2009); Le Pallec and Dupuy-Chessa (2013) or complement the use of Moody's principles with polls Figl et al. (2010) also, thus allowing end-user feedback and involvement during the design process of a visual notation. However, the previous works are usually centered to specific DSMLs and do not provide mechanisms to be calculated to any DSML as our approach addresses. Other works such as Kahraman and Bilgen (2013) propose an evaluation framework focused on language features and therefore not particularly analyzing the quality from an end-user perspective. To the best of our knowledge, ours is the first proposal to generically assess the cognitive quality of DSMLs under development.

Finally, the representation of the collaboration rationale is related to the area of requirements negotiation, argumentation and justification approaches such as Jureta et al. (2008). The decision algorithms proposed in those works could be integrated in our decision engine. Other decision engines such as CASLO Padrón et al. (2005) or HERMES Karacapilidis and Papadias (2001) could also be used.

CONCLUSIONS

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 or in the creation of new models. 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 definition or use of the modeling language. The approach provides two front-ends (i.e., Eclipse-based and web-based ones) to facilitate its usage and also incorporates a recommender system which checks the quality of the DSML under development.

Once the community reaches an agreement on the language features, our Collaboro model can be used as input to language workbenches in order to automatically create the DSL tooling (i.e., editors, parsers, palettes, repositories, etc.) needed to start using the language in practice. For instance, this would involve automatically creating the configuration files required for XText (for textual languages) or GMF (for graphical ones) from our notation and abstract syntax models.

As further work, we would also like to explore how to support the collaborative definition of the well-formed rules (e.g., OCL constraints) for the DSML under development. As these rules are normally expressed by using a (semi)formal textual language (like OCL), the challenge is how to discuss them in a way that non-technical experts can understand and participate. Finally, we are also exploring how to better encourage end-user participation (e.g., by applying gamification techniques) to make sure the process is as plural as possible.

[11] http://www.modelversioning.org

REFERENCES

Aguilera, D., Gómez, C., and Olivé, A. (2012). A Method for the Definition and Treatment of Conceptual Schema Quality Issues. In *International Conference on Conceptual Modeling*, volume 7632, pages 501–514.

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A Survey on Model Versioning Approaches. *International Journal of Web Information Systems*, 5(3):271–304.

Barišić, A., Amaral, V., Goulão, M., and Barroca, B. (2012). Evaluating the Usability of Domain-Specific Languages. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pages 386–407.

Blackwell, A. F., Britton, C., Cox, A. L., Green, T. R. G., Gurr, C. A., Kadoda, G. F., Kutar, M., Loomes, M., Nehaniv, C. L., Petre, M., Roast, C., Roe, C., Wong, A., and Young, R. M. (2001). Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *International Conference on Cognitive Technology*, pages 325–341.

Brosch, P., Seidl, M., Wieland, K., and Wimmer, M. (2009). We can Work it out: Collaborative Conflict Resolution in Model Versioning. In *European Conference on Computer Supported Cooperative Work*, pages 207–214.

Cabot, J. and Wilson, G. (2009). Tools for Teams: A Survey of Web-Based Software Project Portals. *Dr. Dobbs*.

Cánovas Izquierdo, J. L. and Cabot, J. (2013). Enabling the Collaborative Definition of DSMLs. In *International Conference on Advanced Information Systems Engineering*, pages 272–287.

Cánovas Izquierdo, J. L. and Cabot, J. (2015). Enabling the Definition and Enforcement of Governance Rules in Open Source Systems. In *International Conference on Software Engineering*, pages 505–514.

Cánovas Izquierdo, J. L., Cabot, J., López-Fernández, J. J., Sánchez Cuadrado, J., Guerra, E., and de Lara, J. (2013). Engaging End-Users in the Collaborative Development of Domain-Specific Modelling Languages. In *International Conference on Cooperative Design, Visualization, and Engineering*, pages 101–110.

Cho, H. and Gray, J. (2011). Design Patterns for Metamodels. In *Conference on Systems, Programming, and Applications: Software for Humanity - Colocated Workshop*, pages 25–32.

Cho, H., Gray, J., and Syriani, E. (2012). Creating Visual Domain-Specific Modeling Languages from End-User Demonstration. In *International Workshop on Modeling in Software Engineering*, pages 29–35.

Crepinšek, M., Kosar, T., Mernik, M., Cervelle, J., Forax, R., and Roussel, G. (2010). On Automata and Language Based Grammar Metrics. *Computer Science and Information Systems*, 7(2):309–329.

Dullemond, K., van Gameren, B., and van Solingen, R. (2014). Collaboration Spaces for Virtual Software Teams. *IEEE Software*, 31(6):47–53.

Figl, K., Derntl, M., Rodríguez, M. C., and Botturi, L. (2010). Cognitive Effectiveness of Visual Instructional Design Languages. *Journal of Visual Languages and Computing*, 21(6):359–373.

Gabriel, P., Goulão, M., and Amaral, V. (2010). Do Software Languages Engineers Evaluate their Languages? In *Congreso Iberoamericano en Software Engineering*, pages 149–162.

Gallardo, J., Bravo, C., and Redondo, M. A. (2011). A Model-Driven Development Method for Collaborative Modeling Tools. *Journal of Network and Computer Applications*.

Gallardo, J., Bravo, C., Redondo, M. A., and de Lara, J. (2012). Modeling Collaboration Protocols for Collaborative Modeling Tools: Experiences and Applications. *Journal of Visual Languages and Computing*, pages 1–14.

Genon, N., Amyot, D., and Heymans, P. (2011a). Analysing the Cognitive Effectiveness of the UCM Visual Notation. In *International Workshop on System Analysis and Modeling*, pages 221–240.

Genon, N., Heymans, P., and Amyot, D. (2011b). Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation. In *International Conference on Software Language Engineering*, pages 377–396.

Green, T. R. G. (1989). Cognitive Dimensions of Notations. In Sutcliffe, A. and Macaulay, L., editors, *People and Computers V*, pages 443–460.

Green, T. R. G. and Petre, M. (1996). Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages and Computing*, 7(2):131 – 174.

Grundy, J. C., Hosking, J., Li, K. N., Ali, N. M., Huh, J., and Li, R. L. (2013). Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications. *IEEE Transactions on Software Engineering*, 39(4):487–515.

Hatton, L. and van Genuchten, M. (2012). Early Design Decisions. *IEEE Software*, 29(1):87–89.
Hess, J., Offenberg, S., and Pipek, V. (2008). Community Driven Development as Participation?: Involving User Communities in a Software Design Process. In *Conference on Participatory Design*, pages 31–40.
Hildenbrand, T., Rothlauf, F., Geisser, M., Heinzl, A., and Kude, T. (2008). Approaches to Collaborative Software Development. In *Conference on Complex, Intelligent and Software Intensive Systems*, pages 523–528.
Hussain, Z., Slany, W., and Holzinger, A. (2009). Current State of Agile User-centered Design: A Survey. In *Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society - HCI and Usability for e-Inclusion*, volume 5889, pages 416–427.
Javed, F., Mernik, M., Gray, J., and Bryant, B. R. (2008). MARS: A metamodel recovery system using grammar inference. *Information and Software Technology*, 50(9-10):948–968.
Jureta, I., Faulkner, S., and Schobbens, P. (2008). Clear Justification of Modeling Decisions for Goal-oriented Requirements Engineering. *Requirements Engineering*, 13(2):87–115.
Kahraman, G. and Bilgen, S. (2013). A framework for qualitative assessment of domain-specific languages. *Software and System Modeling*, pages 1–22.
Karacapilidis, N. I. and Papadias, D. (2001). Computer Supported Argumentation and Collaborative Decision Making: The HERMES System. *Information Systems*, 26(4):259–277.
Kelly, S. and Pohjonen, R. (2009). Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22 –29.
Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison Wesley.
Kuhrmann, M. (2011). User Assistance during Domain-specific Language Design. In *FlexiTools workshop*.
Lanubile, F., Ebert, C., Prikladnicki, R., and Vizcaíno, A. (2010). Collaboration Tools for Global Software Engineering. *IEEE Software*, 27(2):52–55.
Le Pallec, X. and Dupuy-Chessa, S. (2013). Support for Quality Metrics in Metamodelling. In *Workshop on Graphical Modeling Language Development*, pages 23–31.
Leenheer, P. D. (2009). *On community-based ontology evolution*. PhD thesis.
Liu, Q., Gray, J., Mernik, M., and Bryant, B. R. (2012). Application of Metamodel Inference with Large-Scale Metamodels. *International Journal of Software and Informatics*, 6(2):1–31.
López-Fernández, J. J., Sánchez Cuadrado, J., Guerra, E., and De Lara, J. (2013). Example-driven meta-model development. *Softw. Syst. Mod.*
Mernik, M., Heering, J., and Sloane, A. M. (2005). When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344.
Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63:81–87.
Moody, D. L. (2009). The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779.
Moody, D. L. and Hillegersberg, J. v. (2009). Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams. In *Conference on Software Language Engineering*, pages 16–34.
Mylopoulos, J., Chung, L., and Yu, E. S. K. (1999). From Object-Oriented to Goal-Oriented Requirements Analysis. *Communications of the ACM*, 42(1):31–37.
Norman, D. A. and Draper, S. W. (1986). *User Centered System Design: New Perspectives on Human-computer Interaction*. Erlbaum, H.
Object Management Group (OMG) (2014a). Model-Driven Architecture (MDA) Specification. http://www.omg.org/mda/specs.htm (accessed on 06/05/2016).
Object Management Group (OMG) (2014b). Object Constraint Language (OCL) Specification. Version 2.4. http://www.omg.org/spec/OCL (accessed on 06/05/2016).
Object Management Group (OMG) (2015a). Diagram Definition (DD) Specification. Version 1.1. http://www.omg.org/spec/DD (accessed on 06/05/2016).
Object Management Group (OMG) (2015b). Meta Object Facility Core (MOF) Specification. Version 2.5. http://www.omg.org/spec/MOF/2.5 (accessed on 06/05/2016).
Padrón, C. L., Dodero, J. M., and Lanchas, J. (2005). CASLO: Collaborative Annotation Service for Learning Objects. *Learning Technology Newsletter*, 7(2):2–6.

Power, J. F. and Malloy, B. A. (2004). A Metrics Suite for Grammar-based Software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426.

Prinz, A., Scheidgen, M., and Tveit, M. S. (2007). A Model-Based Standard for SDL. In *International SDL Forum*, pages 1–18.

Rittgen, P. (2008). COMA: A Tool for Collaborative Modeling. In *Forum at the International Conference on Advanced Information Systems Engineering*, pages 61–64.

Rooksby, J. and Ikeya, N. (2012). Collaboration in Formative Design: Working Together. *IEEE Software*, 29(1):56–60.

Sánchez Cuadrado, J., de Lara, J., and Guerra, E. (2012). Bottom-up Meta-Modelling: an Interactive Approach. In *Conference on Model Driven Engineering Languages and Systems*, pages 1–17.

Sánchez Cuadrado, J. and García Molina, J. (2007). Building Domain-specific Languages for Model-driven Development. *IEEE software*, 24(5):48–55.

Scheidgen, M. (2008). Textual Modelling Embedded into Graphical Modelling. In *European Conference on Model Driven Architecture - Foundations and Applications*, volume 5095, pages 153–168.

Siorpaes, K. (2007). Lightweight Community-Driven Ontology Evolution. In *International Semantic Web Conference*, number 4, pages 951–955.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison Wesley.

SVG (2011). Scalable Vector Graphics 1.1. http://www.w3.org/TR/SVG/.

Tamburri, D. a., Lago, P., and Vliet, H. V. (2013). Organizational social structures for software engineering. *ACM Computing Surveys*, 46(1):1–35.

Troyer, O. D. and Leune, C. J. (1998). WSDM: a User Centered Design Method for Web Sites. *Computer Networks*, 30(1-7):85–94.

Völter, M. (2011). MD*/DSL Best Practices. http://voelter.de/data/pub/DSLBestPractices-2011Update.pdf (accessed on 06/05/2016).

Whitehead, J. (2007). Collaboration in Software Engineering: A Roadmap. In *Workshop on the Future of Software Engineering*, pages 214–225.