Community-Driven Language Development

Javier Cánovas, Jordi Cabot

Community-Driven Language Development

Javier Luis Cánovas Izquierdo, Jordi Cabot
AtlanMod, École des Mines de Nantes – INRIA – LINA
Nantes, France
{*javier.canovas,jordi.cabot*}*@inria.fr*

***Abstract*—Software development processes are becoming more collaborative, trying to integrate end-users as much as possible. The idea is to advance towards a community-driven process where all actors (both technical and non-technical) work together to ensure that the system-to-be will satisfy all expectations. This seems specially appropriate in the field of Domain-Specific Languages (DSLs) typically designed to facilitate the development of software for a particular domain. DSLs offer constructs closer to the vocabulary of the domain which simplifies the adoption of the DSL by end-users. Interestingly enough, the development of DSLs is not a collaborative process itself. In this sense, the goal of this paper is to propose a collaborative infrastructure for the development of DSLs where end-users have a direct and active participation in the evolution of the language. This infrastructure is based on *Collaboro*, a DSL to represent change proposals, possible solutions and comments arisen during the development and evolution of a language.**

I. Introduction

Software development involves the collaboration of many types of participants, including to some extent the future users of the software. While some effort has been put into studying how to make the process more efficient by analyzing the way developers collaborate with each other (e.g., Global Software Development [1]), the role of the users has been mostly neglected. Users are mainly involved during the requirement elicitation and testing phases, and have little to none participation in the actual development phases. This usually leads to software that does not satisfy the customer needs. As a response to such problem, software development processes are increasingly becoming more collaborative, trying to engage users in all development phases [2], [3], [4], [5].

Promoting collaboration is especially appropriate when developing Domain-Specific Languages (DSLs). DSLs offer constructs closer to the vocabulary of the domain. Therefore, DSLs help to face the problem of building software for a particular domain mainly due to their ability to specify easily the aim or intention of the application [6]. When using DSLs, users do not need to learn new technical concepts and can just express their needs using the same concepts existing in their domain. Many approaches and recommendations to develop DSLs have been presented [7], [8], [9], most of them focused on defining either the steps to follow in the creation of a new DSLs or the tips to take into account. However, according to these works and in a similar way as what happens in software development processes, current DSL development processes are usually centered on developers rather than the users. It turns out that even if a DSL is a language specific for a domain, domain experts have very limited participation in its creation.

To improve this situation, we propose to make the development process for DSLs more community-aware, meaning that the process is aware of all the stakeholders involved (i.e., technical and domain expert users). In general, it is well-known that communication, coordination and collaboration (i.e., social activity) between community members is a good sign to create high quality software [3]. Our aim is to turn the DSL development process into a more democratic process that includes at every phase the suggestions of the user community for which the DSL is created. For this purpose, it is important to make easier the collaboration between developers and users, trying to overcome the involved technical issues to facilitate users to participate into the language development process. In a similar way to how many FOSS products are developed (e.g., [10] describes the collaborative process for developing Apache based on a mail-based voting system, whereas [11] describes the assignment of development tasks in the Mozilla project), in our DSL development process, community members have the chance to discuss about language changes and decide which ones should be incorporated, thus improving the effectiveness of the process and the quality of the DSL. A key element in the process is the ability to track the changes and the discussions behind them, providing a clear traceability among the elements of different versions of the DSL so that it can be easily justified why some concepts were created or deleted during the DSL evolution.

Our solution is based on a new DSL to represent the collaborations which arise among the members of a language community. This DSL, called *Collaboro*, allows representing change proposals, solutions and comments discussed during the development and evolution of a language.

Collaboro is implemented as an Eclipse-based tool that can be used by all kinds of users willing to have an active role in the evolution of their DSL. The tool includes a simple decision engine to transform these discussions into actual decisions based on the community agreement. This engine

is extensible to allow for more complex decision procedures when only partial agreements are reached.

The paper is organized as follows. Section II firstly defines the concept of community and presents how to make current language development processes community-aware. Section III describes our approach and it is then contrasted with existing related work in Section IV. Section V ends the paper and presents future work.

II. Towards a community-driven development process for DSLs

In this section we will show how we propose to transform traditional language development processes into community-aware ones. We call *community* to the group of users of the DSL, where by users we mean both the (1) technical level users (i.e., the language developers) and (2) domain expert users (i.e., the end-users of the language). These categories may be overlapping, especially when the DSL is a technical DSL (e.g., if the DSL is aimed to write configuration files then the domain experts may be also technical-savy enough to create the language themselves). In any case, the collaboration needs in both cases are the same.

The specification of a DSL involves three main components [12]: abstract syntax, concrete syntax, and semantics. The abstract syntax defines both the language concepts and their relationships, and also includes the rules constraining the models that can be created. The concrete syntax defines a notation (textual, graphical or hybrid) for the abstract syntax, and a translational approach is normally used to provide semantics. As a first approach, we will focus on the community-driven language development of the abstract syntax of a language, which is usually expressed by a metamodel.

According to [7], a DSL is built following a development process composed of five phases, namely: decision, analysis, design, implemementation and deployment. The decision phase allows identifying the need of creating a DSL for a particular domain. In the analysis phase, the problem domain is analyzed and domain knowledge is gathered. In the design and implementation phases the new language is created, which is finally released in the deployment phase.

In a traditional language development process, domain experts participate in the first two phases but then they do not see how their input has been interpreted until the deployment phase. The validation process is therefore performed at the very end of the development process, when bugs and other problems derived from misunderstandings during the first phases can be identified. This situation usually forces to restart the development process to fix the detected problems, thus increasing the cost and effort to create the language. Figure 1a shows the phases in which end-users can participate in a traditional process and illustrates how a disagreement in the last phase involves restarting again the process.

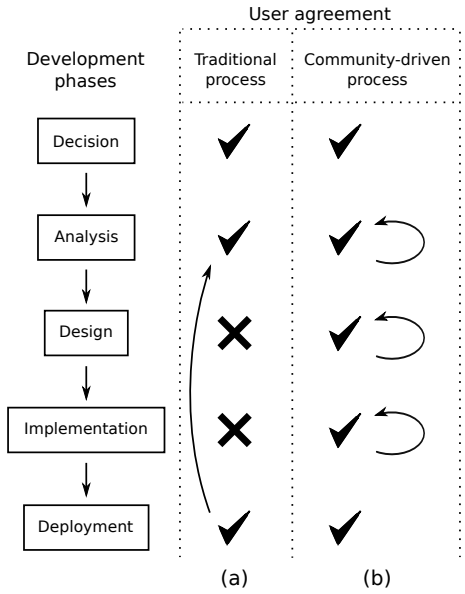

Figure 1. End-user agreement in (a) a traditional language development process, where a restart may be required due to misunderstandings, and in (b) a community-driven process, where each development phase is agreed by the community, thus involving a possible restart in the same phase.

However, when the language development process is made community-aware, both developers and end-users collaborate in all the development phases, particularly, the design and implementation phases which were ignored before. This new commnunity-driven orientation avoids waiting until the end of the process to perform the validation. Each development phase is therefore validated as it is performed so that all parties are ensured that what is being developed will satisfy their expectations (see Figure 1b).

To make this development process feasible, we need to provide adequate tool support for the proposals and discussions around the language development. In the following we show the difference in both approaches by means of an example and illustrate the kind of collaboration information we need to record to enable our new community-driven process. Next section will present our approach for providing such collaborative infrastructure.

Our running example is based on the development of a DSL for production systems (Figure 2). This example DSL is aimed at chief production officers that need to plan the best organization for the production lines for their factory. To this purpose, the DSL offers as constructs concepts like operator, machine, piece, etc., thus allowing the representation of specific product manufacturing settings.

Once the community agrees on the need of creating this DSL in the decision phase, the design phase begins with the abstract syntax definition. Figure 2a shows a tentative version of the abstract syntax metamodel of the language.

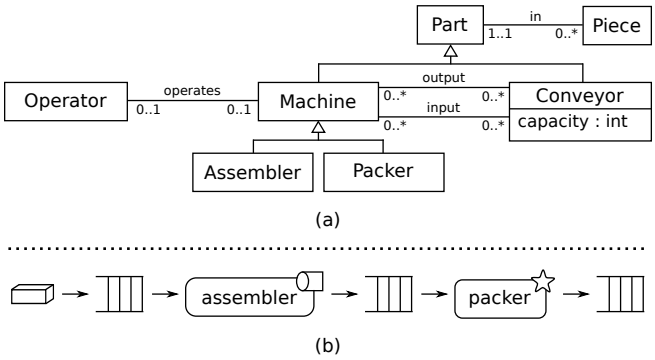

Figure 2. (a) First version of Production System metamodel. (b) Example of a Production System model using a possible graphical syntax.

The DSL allows representing the parts (`Part` metaclass) which a system production is composed of, namely, machines (`Machine` metaclass) and conveyors (`Conveyor` metaclass). A system can be composed of two types of machines: assemblers (`Assembler` metaclass) and packers (`Packer` metaclass). A machine is connected to others by means of conveyors (`Conveyor` metaclass) and the products are represented as pieces (`Piece` metaclass). Moreover, a machine is controlled by a human operator (`Operator` metaclass). Finally, the capacity (`Capacity` attribute) of a conveyor indicates the number of products that it can carry.

Next, in a traditional language development process, developers would define the concrete syntax (see an example graphical syntax for the DSL in Figure 2b) as well as the corresponding tooling (e.g., a model editor for the DSL). Only at the end, end-users have the chance to review the DLS and detect possible errors. For instance, imagine that end-users want to be able to specify the maintenance condition of each conveyor (e.g., good, fine, rusty, etc). Since this is not covered by the metamodel, the abstract syntax has to be updated but this triggers a change on the concrete syntax and the tooling as well.

Instead, in a community-driven process, this missing feature can be detected just after the abstract syntax is provided and, more importantly, end-users not only can propose a change request to incorporate it but also can propose solutions, give their opinion on the solutions presented by the language designers and eventually decide altogether which one to select. Even if end-users may be not technical, the fact that they are discussing about developing a DSL facilitates they can take part in the discussion since the vocabulary they need to employ is the same they use in their daily activities.

An example of such collaboration scenario for this new *condition* feature could be the following:

(a) *End-User 1* realizes that the current version of the language does not allow specifying the condition of conveyors. In the community, the condition is usually measured according to some values considered *de facto*: superb, good, fine and old. Thereby, *end-user 1* proposes to change the abstract syntax metamodel to support such feature.

(b) The change proposal is accepted by the community (i.e., after discussion, it is considered a valuable addition to the language).

(c) *Developer 1* implements a solution adding to the `Conveyor` metaclass an string-based attribute called `condition` (Figure 3a shows the solution developed).

(d) *Developer 2* argues about the correctness of the solution and comments that the type of the `condition` attribute should be enumerated and its possible values should be the ones used by the community to describe the condition of a conveyor.

(e) The community discusses and finally agrees with what the *developer 2* commented.

(f) *Developer 1* changes the solution, thus creating the abstract syntax metamodel shown in Figure 3b, which incorporates the comment.

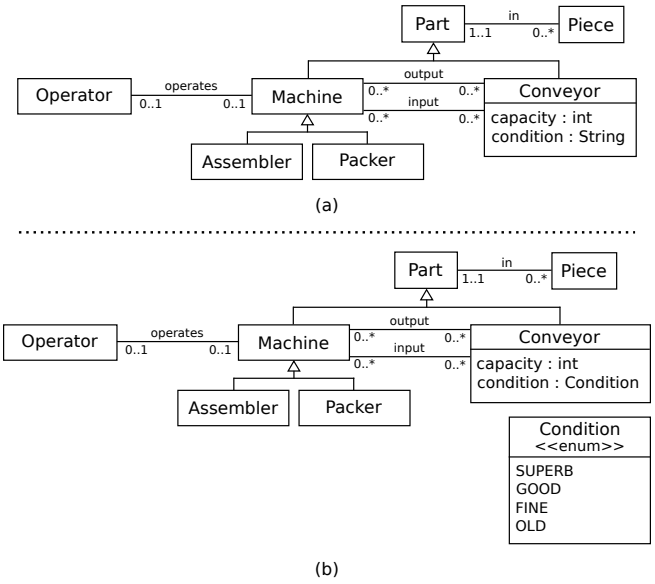

Figure 3. (a) First solution proposal for supporting conveyor condition in the Production System metamodel. (b) Final solution proposal.

Once the community has reached an agreement for the change proposal and solution, they are incorporated into the abstract syntax of the language, creating a new version of the language. Moreover, the proposal and solutions are recorded, thus keeping a track of every change performed in the language. Therefore, at any moment, we can query this traceability information to discover the rationale behind the metamodel elements of the language.

Only when there are no more change requests for the abstract syntax, developers start with the definition of the concrete syntax. A collaboration process to improve also the notation of the DSL could follow a similar procedure to the one described herein for the abstract syntax but a complete support for this is out of the scope of this paper.

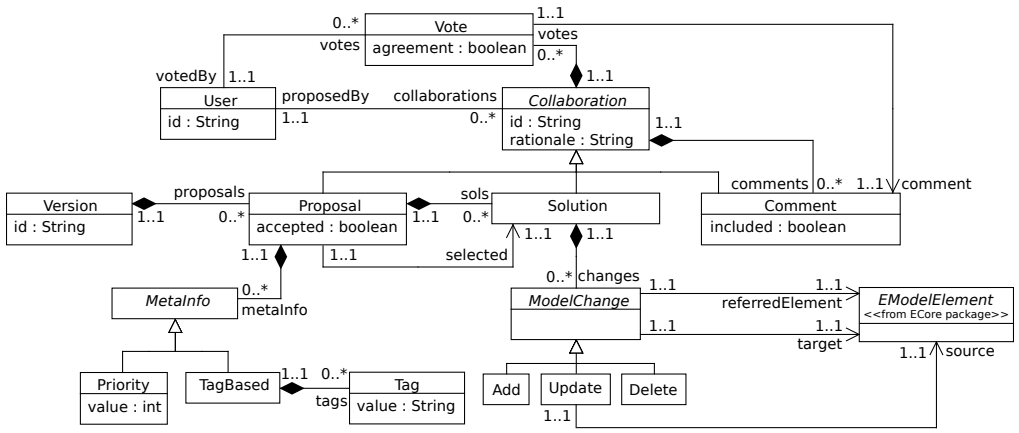

Figure 4. The Collaboro metamodel.

III. *Collaboro*: A DSL TO REPRESENT COMMUNITY COLLABORATIONS

Our proposal for a community-driven language development process is built on top of *Collaboro*, a new DSL that enables the explicit representation of the collaborations that take place during the language development process. These collaborations, expressed as Collaboro models, are then used to decide (and track) the changes to be applied to the DSL under development/evolution.

The abstract syntax for Collaboro is shown in Figure 4, whose development was performed collaboratively in the research team. The metamodel stores both static (e.g., change proposals) and dynamic (e.g., voting) aspects of the collaboration.

A. Static part

Language evolution results in different versions (`Version` metaclass) of the language[1]. 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), which are linked to the parent collaboration they are expanding on. A collaboration is proposed by a user (`proposedBy` reference) and includes an explanation (`rationale` attribute).

The accepted solutions for a set of change proposals are integrated into a new version of the language. The changes to perform are part of the solution proposal. Each solution involves a set of add/update/delete changes on the abstract syntax of the DSL (`ModelChange` metaclass and subtype metaclasses). `ModelChange` links the collaboration infrastructure with the DSL under discussion. In particular, `ModelChange` 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 (i.e., a new version for an attribute). The additional `source` attribute indicates the element to be updated (i.e., the attribute which is being updated).

[1]We plan to support the concept of *branch* in future versions.

B. Dynamic part

Additional metaclasses keep track of the decision process. Collaboration elements are voted by the community, thus allowing to reach agreements. The vote for a collaboration (`Vote` metaclass) represents if the user (`votedBy` reference) agrees or not with it (`agreement` attribute). Thereby, a vote is added to a Collaboro definition when the community member exercises the right to vote.

When an user votes against a collaboration, he/she should include a comment arguing his/her decision (`comment` reference of `Vote` metaclass). The community can then also vote the comment itself. The refinement of the collaboration will eventually be made by the proponent of the voted proposal/solution, who decides to take the comment into account (the `included` attribute of `Comment` metaclass records this fact) according to its voting information.

Proposals can be accepted or not, meaning that the community agrees that the requested change is necessary (`accepted` attribute). For each proposal we can have many possible solutions but in the end one of them will be selected (`selected` reference of the `Proposal` metaclass).

Part of this data (like the `accepted` and `selected` properties) is automatically filled by the decision engine in charge of analyzing and resolving the collaboration, which we will present in the next section.

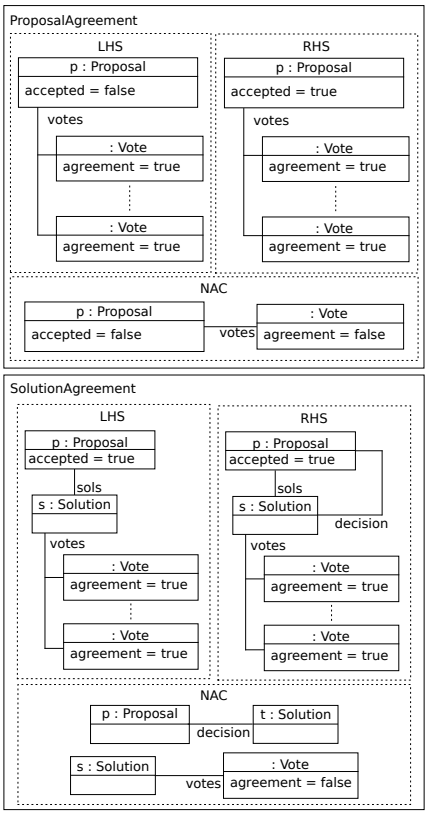

Figure 5. Rules applied in the decision engine using graph-based model transformation notation.

C. Decision engine

The abstract syntax provides the schema to store all the information regarding the collaboration process. The decisions (i.e., approval of change proposals and selection of solutions) coming out of such collaborations can be made by a community manager or could be done automatically by a decision engine following a predefined resolution strategy (e.g., unanimous agreement, majority agreement, etc).

As an example, Collaboro implements now a simple strategy based on a total agreement among the members, i.e., the decision engine integrated in our tool applies the following decision rules (expressed as graph transformation rules [13] in Figure 5): i) a proposal is accepted if there are only positive votes, that is, all users agree with the requested change (see rule *ProposalAgreement*), and similarly, ii) a solution is accepted if all votes are positive (see rule *SolutionAgreement*).

D. Example

To illustrate the language, we will show how Collaboro can be used to support the collaboration scenario described in Section II.

Figure 6 shows the Collaboro model corresponding to this collaboration. The figure is divided in several parts according to the collaboration steps described in Section II. For the sake of clarity, the references to `User` metaclass instances have been represented as attributes and the `rationale` attribute is not shown.

Figure 6a shows the Collaboro model just after *end-user 1* requests to support the definition of a *condition* status for conveyors. It includes a new proposal instance whose `id` attribute is `p1`. The rationale of the proposal is *To better assess the condition of the system, we need to be able to specify the condition of the conveyors, usually classified as superb, good, fine and old*. 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 Figure 6b). Then, a new solution is proposed by *developer 1* (see the `Solution` instance in Figure 6c), which involves enriching the `Conveyor` metaclass with a string-based attribute.

However, this solution is not accepted by all the community members: when voting such solution, *developer 2* does not agree and explains his disagreement with the comment *This type of information is usually represented by enumerates, particularly when the values are known* (see Figure 6d). Since the comment is accepted (see Figure 6e), *developer 1* decides to update the solution to incorporate the community recommendations (see Figure 6f). It is important to note that the elements describing the model changes in Figures 6c and 6f are mutually exclusive (i.e., 6f is an evolution of 6c once the community agrees that the comment from *developer 2* must be taken into account). Moreover, the attribute `included` of the `Comment` element in Figure 6d 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 `decision` reference is initialized with the `Solution` instance). Now the development team can modify the DSL 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 Collaboro model, which will allow community members to know why both the `Conveyor` metaclass was changed and the `Condition` enumerate was added.

E. Implementation

Collaboro is available as an Eclipse plugin[2]. Current version works with the EMF framework and therefore allows the community-driven development of Ecore models. The tool provides a set of views and editors seamlessly integrated in Eclipse, thus facilitating community members to create proposals, solutions and comments from within the same environment they use when using the DSL. These views/editors

[2]http://code.google.com/a/eclipselabs.org/p/collaboro/

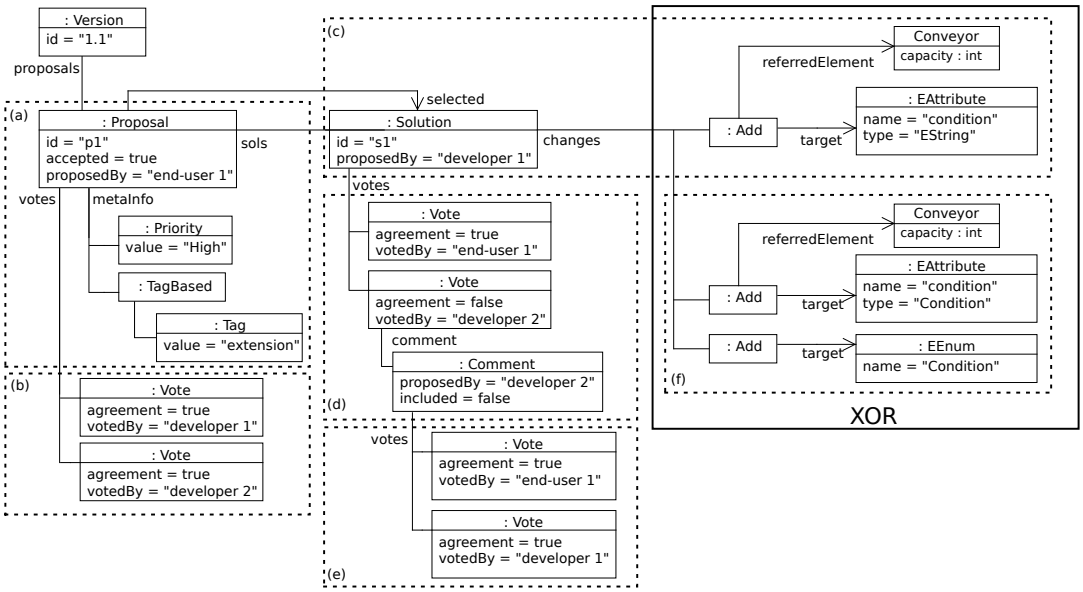

Figure 6. The collaborations arisen in the production system example represented by Collaboro.

can be considered as a kind of concrete syntax of Collaboro since through them members can manage Collaboro models. Figure 7a includes a snapshot of the environment showing the last step of the collaboration described previously. As mentioned above, the tool also includes a decision engine to infer community agreements from the voting information on proposals and solutions.

Figure 7b summarizes the collaboration process. Firstly, community members use the provided Eclipse views/editors to define and discuss about language changes (see step 1). A Collaboro model is kept synchronized with the views/editors as the collaboration is running. Afterwards, the decision engine analyzes the current Collaboro model and derives a new Collaboro model containing the proposals/solutions agreed by the community (see step 2), which automatically updates the individual views (see step 3). So far, our tool does not actually perform the agreed changes on the target DSL. This is still responsibility of the language designers. In the future, we plan to integrate our tool with model versioning tools to automate this step as well.

IV. Related work

Promoting collaboration is currently being taken into account by methods (e.g., user-centered methods such as agile-based ones) and development projects, especially in the context of FOSS communities [10], [11]. [14] introduces the concept of *community-driven development* in the development of a software product. However, they are not aimed to enable community collaborations in DSL development processes. Other works [3], [1], [4] comment on making more participative some model-based phases of the development process, but they do not present the collaboration as a process of discussion and argumentation in a community as ours does nor they provide an actual infrastructure to enable the collaboration.

Regarding specific subsets of our proposal, the model-based definition of metamodel changes is also a topic of interest for model versioning tools such as [15], [16]. Collaboro has been inspired by these tools to express the solutions (i.e., changes to be made) for the proposals. However, Collaboro offers a more expressive discussion environment, such as giving support to the discussion phase and storing the rationale behind each change.

Online modeling collaboration tools [17], [18] allow developers to discuss changes in a synchronous way. Instead, Collaboro enables offline collaborations and a more formal representation of the collaborations (e.g., voting system, explicit argumentation and rationale, traceability).

The incorporation of rationale to community members collaborations is related to requirements negotiation, argumentation and justification approaches such as [19]. These approaches allows applying decision algorithms to arguments in order to infer a justification. Collaboro could be extended to integrate and apply such algorithms.

V. Conclusion and future work

In this paper we present Collaboro, an approach to enable the participation of the user community in the development of a DSL. Collaboro allows representing language change requests and solution proposals (as well as comments to both). Collaboro is available as an Eclipse plugin.

As further work, we would like to apply Collaboro not only to support the collaborative development of the language abstract syntax but also that of its concrete notation. We also plan to advance towards a “change by example”

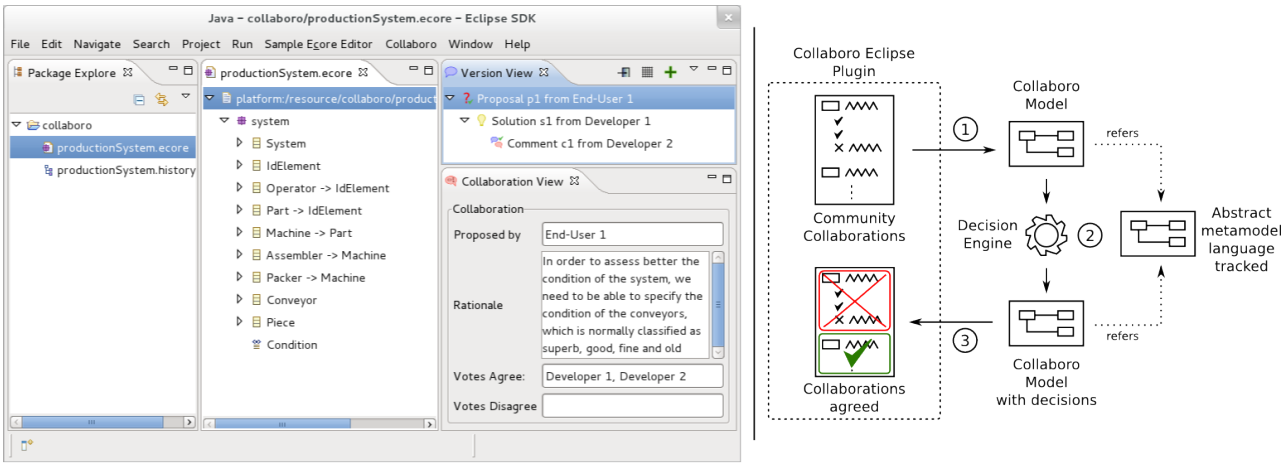

Figure 7. (a) Snapshot of the Collaboro Eclipse plugin. (b) Collaboro process.

approach where end-users can suggest changes by providing example models (possibly inconsistent with the current DSL version) of how they would like to represent certain scenarios. Finally, we will work on extending the decision engine to support more complex resolution algorithms. To this aim, we plan to study works based on ontologies [20] and folksonomies [21], [22] for the automatic inference of relevant knowledge for the resolution.

References

[1] F. Lanubile, C. Ebert, R. Prikladnicki, and A. Vizcaino, "Collaboration tools for global software engineering," *IEEE Software*, vol. 27, no. 2, pp. 52–55, 2010.

[2] G. Booch and A. W. Brown, "Collaborative development environments," *Advances in Computers*, vol. 59, pp. 1–27, 2003.

[3] T. Hildenbrand, F. Rothlauf, M. Geisser, A. Heinzl, and T. Kude, "Approaches to collaborative software development," in *Conf. on Complex, Intelligent and Software Intensive Systems*, 2008, pp. 523–528.

[4] J. Whitehead, "Collaboration in software engineering: A roadmap," in *Future of Software Engineering*, 2007, pp. 214–225.

[5] Agile Manifesto. http://agilemanifesto.org/.

[6] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems*, vol. 45, no. 3, pp. 621–645, 2006.

[7] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, pp. 316–344, 2005.

[8] S. Kelly and R. Pohjonen, "Worst practices for domain-specific modeling," *IEEE Software*, vol. 26, no. 4, pp. 22 –29, 2009.

[9] M. Völter, "MD*/DSL best practices."

[10] R. T. Fielding, "Shared leadership in the apache project," *Commun. ACM*, vol. 42, pp. 42–43, 1999.

[11] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 309–346, 2002.

[12] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels.* Addison Wesley, 2008.

[13] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[14] J. Hess, S. Offenberg, and V. Pipek, "Community driven development as participation?: involving user communities in a software design process," in *Participatory Design*, 2008, pp. 31–40.

[15] AMOR Repository. http://www.modelversioning.org/.

[16] K. Altmanninger, M. Seidl, and M. Wimmer, "A survey on model versioning approaches." *Web Information Systems*, vol. 5, no. 3, pp. 271–304, 2009.

[17] P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer, "We can work it out: Collaborative conflict resolution in model versioning," in *Conf. on Computer-Supported Cooperative Work*, 2009, pp. 207–214.

[18] J. Gallardo, C. Bravo, and M. A. Redondo, "A model-driven development method for collaborative modeling tools," *Network and Computer Applications*, 2011.

[19] I. Jureta, S. Faulkner, and P.-Y. Schobbens, "Clear justification of modeling decisions for goal-oriented requirements engineering," *Requirements Engineering*, vol. 13, pp. 87–115, 2008.

[20] C. Lange, U. Bojars, T. Groza, J. Breslin, and S. Handschuh, "Expressing argumentative discussions in social media sites," in *Workshop on Social Data on the Web*, 2008.

[21] S. Angeletou, M. Sabou, L. Specia, and E. Motta, "Bridging the gap between folksonomies and the semantic web: An experience report," in *European Conf. on Semantic Web*, 2007, p. 93.

[22] S. Purao, V. Storey, V. Sugumaran, J. Conesa, J. Minguillón, and J. Casas, "Repurposing social tagging data for extraction of domain-level concepts," in *Natural Language Processing and Information Systems*, ser. LNCS, vol. 6716, 2011, pp. 185–192.