Collaborative development of domain-specific languages, models and generators

Juha-Pekka Tolvanen
MetaCase, http://www.metacase.com
jpt@metacase.com
http://twitter.com/mccjpt

Almost all software development activities require collaboration, and developing domain-specific languages is no exception. Language users provide feedback as the language is developed, and also different parts of the language can be developed in parallel: for example, one developer can focus on the abstract syntax, another on the notation, a third on code generators, and a fourth on integration with the development process. This collaboration becomes even more relevant when a number of integrated domain-specific languages are developed. In this article we share our experiences on how teams can collaboratively develop and use domain-specific modeling languages, and what benefits this collaboration provides.

1 Domain-specific languages and collaboration

Domain-Specific Modeling (DSM) has become popular in recent years. This is no surprise given the reported benefits of significantly improved productivity and quality [Sprinkle et al. 2009]. Working on the higher level of abstraction offered by a language, with automatic transformations producing the lower level “implementation”, has been a recipe for success for decades. Not all modeling languages, however, lend themselves to automatic transformation. Languages that do not focus on a specific problem domain – e.g. general purpose languages like UML and SysML – cannot raise the level of abstraction up to the problem domain. Nor can they guarantee that the models created are complete and correct to enable code generation. These general purpose modeling languages are typically used only for sketch models, which are thrown away afterwards [Collins-Cope 2014, Petre 2014]. In contrast, a DSM language focuses on a narrow area of interest and enables executable specifications [Sprinkle et al. 2009]. Code generators provide automation by reading the models created with the language to produce various kinds of artifacts like code, configuration, test data and documentation. This automation is possible because both the language and generators need to fit the requirements of only a single domain – often inside just one company.

Creating domain-specific languages calls for collaboration. First, it is common to distinguish language creation and language use. Second, and related to language creation, the abstract syntax of a language, its rules, notation, generators and tooling is not always created by a single person. It is therefore natural that the work can be shared and the development team can collaborate. Third, while a domain-specific language focuses on a small area of interest, a single language is not always enough. Applications are large, they have connections to other systems, include various sub-domains and different developers and tasks require different views. This again calls for collaboration to provide several distinct yet integrated languages. Finally, tools play an important
role and since some tools require considerable more effort to provide modeling language support than others (for a review see [El Kouhen et al. 2012]) it is natural that the work is shared.

2 Example language for collaborative development and use

In this article we focus on collaborative language development and use. We describe the different ways to collaborate and the benefits they provide. To make things concrete we use an example case from developing heating systems. Here two integrated domain-specific modeling languages are used: one specifies the structure of the system such as various instruments connected via pipes, and the other describes behavior of an instrument (Figure 1). The implementation of these languages is documented in detail and available for modifications [MetaCase A 2014]. Regardless of the tooling applied the language development practices and need for collaboration are naturally universal.

Figure 1. An example of DSM: specifying structure and behavior of a heating application

On the structural part, the diagram on the left shows valves, sensors, a burner, a pump and other instruments along with their pipe connections. All these modeling concepts are also directly the domain concepts. In other words, the language maps closely to the problem domain. The behavior of these instruments is described with another modeling language. The state machine on the right shows an example of this, defining the behavior of pump ‘P1’. In addition to states and transitions the instruments of the heating system are used as conditions and actions. For example, the behavior part of pump ‘P1’ depends on the status of burner ‘HU1 B1’: it is turned on if flame is detected from the burner.

These two DSM languages are integrated as it would not make sense to specify behavior for instruments that are not part of the system structure and vice versa. These two languages also share some of the same concepts, like the burner and pump illustrated above. The domain-specific models are not just pictures, they are formal specifications, their consistency and completeness is checked by the language definition, and most
importantly they can be used to generate fully functional production code directly from the models. The same models can also be used to produce deployment and installation, test data, material calculation, documentation, etc.

3 Collaboration between language engineer and language user

The most typical form of collaboration is between language engineer and language user. In the best case, once any element of the language is defined, language users may immediately test it. An example of such tight collaboration is shown in Figure 2, where the left side illustrates language definition and the right side language use. Within the heating system the language definition on the left shows the concept ‘Sensor’ and its properties along with the definition of its notation. The diagram on the right describes its use while specifying a temperature sensor ‘TS6’ that is installed in the control room to indicate the current temperature in the attached pipe. The model showing the sensor can thus describe just those aspects of sensors that are defined in the language (aka metamodel). Also, sensors can only be used in the manner the metamodel allows, e.g. they need to be connected to a pipe but not to another sensor. The language definition may also include more complex rules, like those related to a kind of sensor: e.g. temperature sensors must have one connection to a pipe, but flow sensors must have two.

Figure 2: Collaborative language definition and language use

In this case the language development practice, and supporting tool, enables working as a pair in an agile manner. Any change in the language definition can be immediately applied by other members of the team. Such a tight collaboration between language definition and its use brings several benefits. Many of these are
common for all user participatory approaches, but particularly relevant as often language engineers do not have prior experience on creating languages.

The benefits of collaboration include:

- Enable early feedback and validation of language definition. Users may immediately test the language and not only verify that the definition is correct, but also validate that the language lets users specify the kinds of things for which it is intended.
- Minimize the risk of creating the wrong language constructs and enable the language to be defined in small increments. This is particularly important if the domain is new, evolving, or the language engineers do not have prior experience of language development.
- Language adoption and acceptance improves since language users are involved early in the language definition.
- Speeds up the move to DSM, since while generators are being developed language users can already start modeling. This is particularly relevant for shorter term projects in which languages are needed quickly. If language definition takes months the projects that would need the language have already ended before the language is available.

This collaboration becomes even more relevant during the language maintenance phase when there is already a substantial amount of work done with the modeling languages. Any change in the language definition can then be immediately checked and reflected against the existing models. Also language users can see the influence of the language modification and can propose suitable policies for model updating. For example, easy parts like renaming of language concepts can be automated so the language notation and semantic rules change when the abstract syntax of the language is changed. Also changes made to the language definition can be reflected automatically to the existing models to update them accordingly. If the language update is such that it cannot automatically be reflected in the models, generators can be made to report on those parts of the model that cannot be updated automatically, and have thus been left for modelers to change.

4 Collaboration while defining the same language

A single person is not necessarily good at defining all parts of the complete modeling solution. It may also be organizationally wise to divide the work among several persons. Perhaps one of the most typical ways to divide the work is between the creation of a modeling language and related generators. Figure 3 illustrates this collaboration. On the left side of Figure 3, a language concept ‘Valve action’ is defined with its properties such as action (e.g. on, off) and valve position (e.g. left open, right open, both open, both closed). On the right side, the generator developer defines a code generator for actions within a state transition and uses the same language construct ‘Valve action’. The properties of ‘Action’ and ‘Valve position’ are here used to produce valve-related actions.
Figure 3. Collaborative development of metamodel and generator

Here the metamodel of the language and the generator are both being defined simultaneously. If the same language is used to create models for different generation needs, several generator developers can access the same metamodel definition. Perhaps the most typical order in which language definition takes place is to first focus on abstract syntax defining what kind of models can be made. This can then be extended with rules for keeping models syntactically correct, complete and consistent while generator development is started. Often in cases where the visual appearance of models is relevant, like in user interfaces, hardware or physical devices, the development of notation may also be delegated to other people than those defining the metamodel.

The benefits of having several people involved in creating the modeling solution are:

- Utilize expertise from different people. While a language engineer usually focuses on the language’s abstract syntax and static semantics, others, including future language users, can define the notation. This also improves acceptance of the language as concrete syntax matters – in particular when starting to work with the new language. Also different generator needs call for different kinds of expertise: while one may focus on generating code in a particular programming language, others can make generators for build scripts or integrate with existing libraries. Generators and scripts can also be implemented by other people to check models, annotate errors, provide guidance during modeling, documentation, etc.

- Language and generators can be checked early while being defined. As in any team work, several people see more than one, and can discuss about language definition and generators as well as test them in collaboration – even using the same jointly developed models.

- Development of languages and generators is sped up: not only because different generators can be developed by different people, but because things like notation and some of the checking rules do not need to be completely ready before making generators. This is important as generator development usually takes more time.
Figure 4: The effort to define a modeling language and the effort to define a generator

Figure 4 illustrates the division of effort by inspecting the time needed to develop a modeling language and time needed to develop a generator. In these eight cases where data was gathered on the effort needed, generator development usually took more time than definition of the language (it’s concepts, constraints and notation). All the above cases focused on creating a code generator for one target only. When developing several generators, e.g. for different target platforms, the effort to develop the second and subsequent generators is usually smaller. The language is then better known and the generator developer has identified good practices to access models, and may reuse parts of the existing generators. For example, at Panasonic the second generator for a different, albeit smaller, target platform took significantly less time than the first one [Safa 2007].

The effort to build generators is naturally tool dependent. If a generator is disconnected from the metamodel, needs to parse temporary models, or uses model-to-model transformations to combine different models, it takes more time to develop than if the generator development tool can access the jointly developed language (as in Figure 3 above). When different parts of the language, such as its abstract syntax, constraints, notation and generators, can be accessed and combined they can be better tested and changes made in one part be more easily traced to other parts. This will tend to lead to a better quality modeling solution.
5 Collaboration while defining several integrated languages

When several domain-specific languages are developed the number of people involved naturally grows too: different people tend to master different parts of the whole system. One language can focus on structures, another on behavior, a third on reusable parts in a library, a fourth on configuration and so on. Figure 5 illustrates the joint development of the two languages in our heating system example. The language on the left, P&I Diagram, is used to define the structure of pipes and instruments of the system. In this language the behavior of pumps can be described with the ‘Heating application’ language. In other words, ‘Pump’ can be specified in detail with another language. This second language, based on state machines, is described in the window on the right. To support the nesting of states, each ‘State’ within the heating application can be specified with another submodel using the same ‘Heating application’ language. Integration among these languages is more detailed than shown in the figure as both languages also share the same concepts, such as some of the instruments. Examples of these two languages used in modeling were illustrated in Figure 1.

Figure 5: Integrated definition of the languages for heating system.

If several domain-specific languages are used there can also be several teams developing different languages. Based on our experience the language development team is usually just one or two people, but the largest language development team I am aware of included over 20 people. Naturally the domain the modeling languages target influences this, e.g. if the intention is to gather and integrate knowledge from different disciplines and tasks (software, mechatronics, requirements, variability, configuration, deployment etc.).

The benefits of collaboration include:

- Languages may reuse common parts and the team can integrate languages based on a shared definition. This allows harmonizing the parts shared among the languages and better modularization.
- Integrated languages cover a richer variety of views or aspects of the system. This is important since otherwise integration would need to be handled by defining and maintaining model-to-model
transformations. Even worst, the resulting model transformations would provide a single, one-way-route only: migrating changes to models that have been subsequently edited is challenging or even impossible. Instead, integrated languages can make model integration easier. Consider a change to Pump ‘P1’ in Figure 1. Because the languages are integrated there is only one ‘P1’, and thus changing its properties in one diagram (and DSL) will change it in the other diagram too. The same applies at the language definition level: changing the definition of the ‘Pump’ concept used in both languages will update it in both languages.

- Shared expertise can be combined. For example in the case of automotive embedded systems, a language called EAST-ADL includes over 15 different sub-languages each covering different subdomains (architecture, safety, error modeling, requirements, variability, hardware etc.). A single person can hardly master them all along with their generators. Therefore for example the EAST-ADL implementation at [MetaCase B 2014] has been defined in collaboration among two language engineers – each focusing on different parts of the whole.

- Integrated languages support the development process. Rather than creating and editing the same kind of information in different phases by different people, integrated languages enable the same information to be shared. For example in our case of heating systems, the initial structure of pipes and instruments might be defined, followed by the specification of the behavior that references those structural elements. Inconsistencies among these two views can be reported, e.g. to check that the behavior definition does not reference instruments that are not specified in the system.

Based on the language creation projects we have been involved in, the best place to start are the stable parts – those domain concepts that are well known and have clear semantics. When defining several languages it is good to identify early those parts that are reusable or enable integration. For example, when the same domain concept is used in several languages, each providing a different view of it, it is quite common that notation and generators related to it have similarities.

Tools naturally have an impact here, as some tools permit only one user at a time whereas others enable simultaneous collaboration. Ideally, several people can define different parts of the language at the same time – and language users can test them at the same time too. Since it is hard to get the single language right in the first place, the challenge to get an integrated language correct is even harder. It is therefore important to create the language and use supporting tools that make language modifications easy.

6 Conclusions

The creation of domain-specific languages, generators and models requires collaboration, but development practices – along with tools – tend to focus on a single developer only. This causes problems when gathering feedback from users, utilizing the expertise spread throughout the team, using several related languages, or reusing concepts defined as part of another language.

We described some typical approaches used to create modeling solutions, and the benefits collaborative development offers. Collaboration between the language developer and language users enables agile definition, in which a language can be defined in small parts and tested immediately by language users in concrete development situations. Tight collaboration enables a fast feedback loop, leading to better quality
languages and user acceptance. Collaboration within the language development team allows the load to be shared and utilizes expertise from all members of the team. This not only leads to better defined DSLs but also to faster deployment. Finally, while DSM focuses on a particular small area of interest, a single language is not always enough. Applications are large, they have various aspects, different developers have different views, and no single modeling language can specify it all. In such cases collaborative development is the only realistic option: Language definitions, generators and notation can be reused, languages can be integrated to support the development process, and different expertise can be combined.

7 Acknowledgements
I would like to thank Saïd Assar, Benoît Combemale, Steven Kelly and Janne Luoma for providing feedback to early version of this work.

8 References


MetaCase A, Heating System Example, 2014, 


