New UX for Participatory Modeling

Steven Kelly
MetaCase
stevek@metacase.com

ABSTRACT

In domain-specific modeling, the models are often built by problem domain experts rather than traditional software engineers. The languages are thus designed with the needs of problem domain users foremost, rather than those of software developers. The tooling too should aim to support these modelers. Keeping the language closer to the problem domain also improves participation by others who are even further from the development world: other experts, managers, and project and company-level stakeholders. While the language fits these users too, the tooling is still aimed at functionality for modelers, and these non-modelers' needs will often be different. In this vision paper, we suggest some ways in which those needs differ, look at various approaches to meet them, and consider the challenges involved. In particular, we consider how to meet the needs of such participatory modeling in an efficient manner in the context of modeling tools made with language workbenches. We offer user-assembled model explorers as a low-hanging fruit option, and investigate questions of their liveness and bidirectionality.

CCS CONCEPTS

• Software and its engineering~Software notations and tools ~Context specific languages~Domain specific languages • Software and its engineering~Software notations and tools ~Context specific languages~Visual languages • Software and its engineering~Software notations and tools~Development frameworks and environments~Integrated and visual development environments • Information systems~Information systems applications~Collaborative and social computing systems and tools

KEYWORDS

Participatory modeling, language workbench, metamodel, domain-specific modeling

ACM Reference format:

Steven Kelly. 2024. New UX for Participatory Modeling. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria. ACM, New York, NY, USA, 5 pages.*

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688562

1 Introduction

In domain-specific modeling (DSM) [1], the models are often built by problem domain experts rather than traditional software engineers. The languages are thus designed with problem domain users' needs foremost, and the tooling is targeted at the tool use cases they encounter while modeling. This gives a far stronger focus on users from the problem domain compared to general purpose modeling languages such as UML, let alone manual coding. It also reduces the conceptual distance between models and participants who are even further from the development world: other experts, managers, and project and company-level stakeholders. This offers more scope for involving these people, who may have rather different needs from the modelers. Being able to better involve all participants contributes to project success: the use of models for team communication has been found to have the greatest influence on productivity [2]. In the early days, involving all participants may also influence the initial acceptance of model-driven development.

With most modelling tools for domain-specific modeling languages being built in language workbenches, the user interface (UI) and user experience [1] (UX) solutions in the resulting modeling tools are often fixed by the workbench. The choices there tend to be aimed at users in the range from software developers to domain expert modelers. Our experience in using domain-specific modeling tools with non-modeling participants shows that they often have different UX needs or desires from the modelers. Some of those needs are relatively constant and recurring, others may appear *ad hoc*.

The most common requests by non-modelers are relatively minor changes, many of which can be accommodated by existing functionality of strong language workbenches [3]. One category is to filter out certain model concepts, a feature often already supported by the resulting modeling tool. Another category is to hide details that are less relevant to the participant, or tweak the visual representation to make it similar to some more familiar tool: these can often be supported by conditional notations.

Adding extra UX solutions beyond these simple cases generally requires orders of magnitude more work than using those offered by the workbench. Even for recurring needs, the expense may mean these needs remain unsatisfied. For *ad hoc* needs, the chance of support is basically non-existent.

In this paper, we want to take a new look at this situation. What kinds of UX have been or might be requested by

[1] We consider UX here to encompass UI, usage patterns it offers, user skills and feelings like frustration or empowerment, etc.

traditionally non-modeling participants, and how could a language workbench make it possible to offer them at a practical cost in terms of time and resources: even to the extent of *ad hoc* addition.

2 Background and Related Research

When building a modeling tool from scratch, a wide range of approaches and user interface solutions could be used. In practice, most domain-specific modeling tools are made using a language workbench. There are relatively few language workbenches that have significant adoption. Taking a broad definition, we can separate them into three categories, giving a few examples for each (*italics* if current adoption levels for DSM use are uncertain; citation where a defining publication exists):

- **Textual**: Xtext, *Spoofax* [4], *Rascal* [5]
- **Graphical**: MetaEdit+ [6], MS DSL Tools [7], Sirius, GEMS
- **Projectional**: MPS, *Intentional* [8], *Whole Platform* [9]

The textual language workbenches focus on text, and thus their UX for model data is limited, generally to monospaced text views with some automatic formatting. The projectional language workbenches aim at offering a wide range of UX solutions, up to hand-coding for each language with Intentional. If we want to offer a range of new UX, beyond what tools have now but somehow well-integrated with it, the textual workbenches offer little scope, and the projectional workbenches rather too much scope. In addition, a 2021 global survey across multiple domains found that graphical workbenches are preferred and used significantly more than the other types [10]. We will thus focus our search on the case of graphical language workbenches; solutions found there are of course possible to apply elsewhere.

Graphical language workbenches tend to offer three main UX areas in the resulting modeling tool:

- **Explorers** *for navigating to models and their elements*
  - Project explorer: workspace, solution, projects
  - Models explorer: by type, containment hierarchy
  - Model explorer: objects, sub-objects, sub-models
- **Editors** *for creating, editing and viewing models*
  - Diagram
  - Table
  - Matrix [11]
- **Property views** *for editing model element details*
  - Property dialog (traditional UI widgets)
  - Property sheet (simple grid with mostly textual display)

Research on these has focused almost exclusively on editors, and in particular diagram editors, with a secondary focus on multiple editors on the same model (multi-tool representational independence, blended modeling) [6][12]. Showing sufficient research value in explorers or property views is perhaps difficult, and often regarded as simply a development task. Yet the widely-adopted tools have a rather extensive range of explorers, and property views that often go significantly beyond what is found in other non-modeling application user interfaces: there is value here.

The explorers that exist tend to be fixed, with little or no per-language customization (beyond automatically adapting to the concepts and icons of the language). Some user-level *ad hoc* customization or choices between various options is available, e.g. sorting by name or type, or displaying the tree by type or by containment. Further enhancement generally requires coding[2].

Property views are often strongly linked to the attributes in the language concepts, and most commonly automatically created — although with more freedom for UI and UX decisions by the language designer than in the explorers. Perhaps correspondingly, there is also less freedom on the user-level.

3 Types of New UX

As there is relatively little data on the needs and desires of new kinds of (non-modeling) participants, and they themselves have little experience to base their suggestions on, we will apply an inverted requirements analysis: look at what may be possible, how it might be applied, and its benefits and challenges.

Explorers are generally composed of standard UI widgets: trees, lists and tabs. The elements are obtained by rather simple fixed queries, and their display is normally a simple text such as an element's type and name, possibly with an icon. The rather extensive range of explorers offered, plus their parameterization and filtering options, certainly covers the main use cases of modelers. See Figure 1 for some examples (clockwise from top left): average, simple, and advanced.

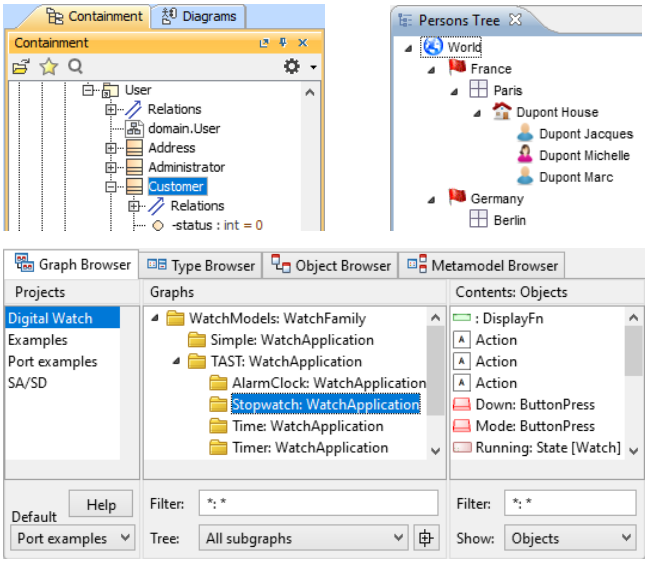

Figure 1: Explorers in MagicDraw, Sirius and MetaEdit+

Explorers' use of only standard UI widgets, along with the simple specification of query and display, mean that low hanging fruit for new UX may be on offer here: new explorers could be relatively easily specified and created by the workbench user or modeler.

[2] https://eclipsesource.com/blogs/tutorials/tutorialshow-to-customize-the-emf-client-platform/

The query would need a starting point (either fixed or chosen at modeling time), a fixed accessor or navigation path returning a list or tree of elements (possibly with filtering and sorting), and optional initial choices for how to display results (icon, type and name versus just name etc.). For some cases, a multi-column output could be useful: an accessor or navigation path per column, starting from that row's element.

For example, the starting point could be the currently open State Model, the query could return a list of all relationships filtered to Transitions and sorted by name, and a second column could show the triggering Event. Such a query could be built up easily, even *ad hoc* incrementally.

The UX for building the query could be based on choices from lists, rather than requiring something more like code. Experience with MetaEdit+ shows at least two similar cases: producing the list of dynamic ports to display in an object's graphical notation [13], and building a model-to-text generator by clicking in lists of the metamodel elements (Transition, Name, and Event) to form query commands (Figure 2).

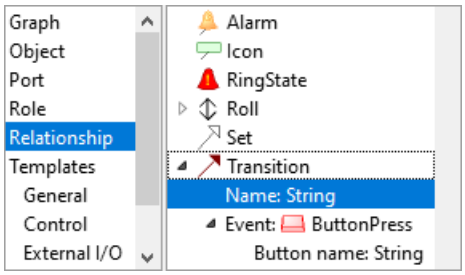

Figure 2: Selecting metamodel elements to build up a query

In terms of the complexity for the query builder, this would be between those two cases. Similar cases are also found in Excel's Pivot Table builder or Power BI, so may also be familiar and practicable even for non-modeling participants.

The output UX could be as a list, tree, table or combination of these (in practice a tree with table columns). Where desired, this could alternatively be displayed as text, with line breaks and tabs as separators. Within the tool, the parts of the text representing objects can still be links to those objects, as in the Live Code of MetaEdit+ generator output.

These new explorers would offer the non-modeler participants a variety of ways to locate model elements, see information of how each is used, and get an overall picture of the elements and their use. This tallies well with our experience of the kinds of information non-modeler participants ask for. The overall picture is something they ask for when trying to understand the models, language and tool in the first place. Finding elements and seeing their use is often requested when helping a modeler.

A further level of functionality could go beyond simple UI widgets and simple navigation commands to offer features more like Power BI and similar reporting tools: what is often called a dashboard. This requires the ability to count, sum and show ratios, weighted averages, and other mathematical and statistical information, and to display output also as business graphics like pie charts and bar charts. These features are probably most useful in project management roles, e.g. to give an idea of the size of projects and models. They are however perhaps better offered outside the modeling tool: existing dashboard tools could query the modeling tool for model data (and perhaps metadata).

4 Liveness of New UX

When adding features to a complex environment like a modeling tool, it is good to consider the use process beyond the immediate result [14]. What do users want to do with the results, how will they interact with them, who else might use them? Are the results static data from a moment in time, fully active parts of the models, or somewhere in between? We can separate these cases depending on whether the results have a bidirectional link to the models, e.g. we could add an object to the result list, and it will be added into the model from which the list was produced. (For simplicity, we will consider the case where the new UX is a simple output list, unless otherwise stated.) We will look first at simpler cases, which can be considered on a scale of increasing liveness:

- Just get the results as text, a list, table, tree etc.
- Be able to keep the result open in the tool, save it, or export to other tools (e.g. Word, Excel).
- Output to Live format, where the elements can be inspected and their links followed in the same way as when the same elements appear in the models themselves, editing them there.
  - This should be simple to achieve for the non-text case.
- Be able to save the query itself, repeat the query to obtain the same results (e.g. later or for another user), or the updated results (if the model has changed in the meantime).
- Automatic update of the result as the models change (re-running). (A user could also choose to stop this.)
- Keeping the same elements and surrounding boilerplate (text or UI representation), but have the elements update as their contents change (e.g. a name change).
  - Is this useful, or would it always be better to re-run, e.g. if the name change would affect the order, or affect whether that element would be included in the result.

4.1 Bidirectionality of New UX

In some cases, users may want to allow a kind of round-trip engineering, where the query results can be edited — beyond simply changing properties in them via the normal tool facilities. In these cases, the results must retain some structure of their own, but also remember the link from that to the model structures that returned the result. We can of course also allow operations that only affect the output list (changing order, removing elements, adding elements). These have proved useful in similar situations, but may carry the risk of the user expecting that those actions take effect on the original models, not just the output list.

Operations on the output list that could have a corresponding effect on the source models include:

- Moving an element to a different position in the list, where the list was not sorted but maintained the order from the source model.
- Removing an element, causing its removal from the model from which the element came.
- Adding an element, causing its addition to the model that forms the context of that query.

For some cases a 1:1 correspondence between the source model and the structure of this query's result is obvious, and supporting these bidirectional operations may thus appear relatively trivial. For such cases, this will still leave interesting questions of how a change in one representation should affect the underlying model data or other representations of the model. We will defer looking at examples of these promising cases to the following subsection.

For many other cases, there are serious questions and difficulties over such bijections between query outputs and models. Is a model in an editor just a projection of the underlying model, and the query result just another projection? Or are some representations afforded a higher status, effectively being the actual model, the format within which it is created and edited, whereas others are simply displays of that model?

Can we make it so that operations in the new UX act on the underlying model, and their effects then apply to the various projections simply by the projection mechanisms themselves? This is of course the premise of the idea of purely projectional editors (somewhat outside our scope here). Projectional editors cannot support representation-specific information, but in graphical language workbenches all forms of representation allow such extra information on top of the actual abstract model data. Some of that extra information may be expressed once by the metamodeler (notation definitions), or once per model representation by the modeler (representation settings), without needing information per representation element. But representation information like the position of an element are important to the human user, who remembers elements by their position, or as a series by rote.

Although an alphabetical list is useful, in many cases a human-ordered list is more useful, at least up to a certain size. Similar factors apply to visual representations: the 2D layout of a diagram is often a vital part of presenting its contents in a form that is easy to assimilate, and the position of an element within the diagram is often one of the strongest things that a modeler will remember – even more so than its name, in many cases. This is only natural: we have been able to navigate our way around the 2D surface of our planet for a much longer time than we have named individual places.

For the non-modeler participant, the situation is less clear. For some of them, the models will be familiar at least in read-only form, or then things like a familiar order will already be found from the domain itself. In other cases, an alphabetical ordering in the new UX is more likely to be useful. But even then, if we want bidirectionality, we need to take into account the representational information in the other representations of models in the normal UX.

4.2 Bidirectionality Example Cases

Allowing editing in the new UX thus raises more complex questions than one might expect. The general problem of round-trip engineering is familiar, but one might have hoped that having both ends of the transformation be models rather than one being plain text would help, as might having a formal specification of the transformation from the model to the new UX.

As a basis for thinking about the use process, we can take a simple practical example case: some Use Case models with Actor objects, and a new UX giving a list of all Actors. It is easy enough to see how operations on an Actor and its simple attributes could be allowed to work in the same way as those operations on the Actor in the normal UX. But what about operations like adding or deleting an Actor in the new UX? If we add an Actor, to which of the Use Case models should it be added? If we delete an Actor, should it be deleted from all Use Case models where it appears? Is there an analogy to deletion within the normal UX, where generally deleting an Actor in a Use Case model only removes it from that model, without affecting its use in other models? In that case, deleting the Actor in the new UX would only remove it from the new UX. Would that then be an instruction to always omit that Actor when refreshing the new UX? That approach is seen in Excel, when editing Pivot Table results; a subsequent problem is however the list of such omitted elements, which is not available for operations in the Excel UI.

Let us take a step back to the simplest possible case: a single bit. For instance, we could have a Boolean that says whether the modeled system is network-enabled. That could be an attribute on the root element of the model, and could map to a single check box in the new UX. Clearly, operations on that bit can easily maintain consistency between the normal UX and new UX. But as soon as we allow the possibility of more than one model, things become less simple. In addition to the checkbox and its state, we need to know which model it is associated with. The new UX must thus maintain the identity of the model, or some query or navigation path that will reliably return the same model later.

What about cases where the new UX is not just a subset of model elements, but built by operations collecting information from more than one model element. E.g. if we specify for each model element whether it is network-enabled, and the new UX shows just one checkbox saying whether any of the model elements are network-enabled. If we turn that checkbox on, we have no way of knowing what the resulting operation in the model should be. But if we turn the checkbox off, we would know to turn network-enablement off in every model element. Such asymmetrical operations are however rarely useful, and the better approach may well be to treat that new UX as read-only.

4.3 Bidirectionality Heuristics, Tool Limitations

In deciding how to deal with bidirectionality in the new UX, we can look at existing similar functionality in the modeling tool. Tools differ widely in their support for multiple representations

and propagation of operations. This constrains our freedom, but also informs our choices.

For instance, in MetaEdit+ a graph may have several diagram representations, and a given object may be represented in more than one of them. Removing the object's representation from one diagram still leaves its representations in the other diagrams, and the object itself is still in the graph. With default settings, the object will be removed from the graph automatically when its last representation is removed from representations of that graph. This seems to offer users the best combination of high functionality, low surprise and no unexpected data loss. It is similar to the approach taken in hard links in file systems.

When applying this information to the question of bidirectionality in the new UX, we need to include the user perception of the new UX. If it is considered more like the results of a query or generation, then the user is unlikely to expect operations on the results to have widespread effects on models. The new UX results are seen as temporary output, unlike the persistent model representations such as diagrams.

The answer may also differ according to the tool; for other tools we can of course only surmise and speak in generalities here. Projectional editors have no persistent representations, but of course have persistent abstract model data and non-persistent transformation outputs. Lacking the indirection of a representation concept, they may however find it hard to cope with treating new UX results as a projection of a persistent model with bidirectional actions. EMF-based editors like Sirius have a forced concept of strong containment, with each element contained by exactly one other element; other references are non-containment. Bidirectional operations could be possible, but may require significant proxying and housekeeping.

5 Conclusions

New kinds of modeling participants may benefit from new kinds of UX, provided as extensions in existing modeling tools made by language workbenches. Some kinds of new UX seem to be eminently possible to create with little effort, in particular explorers for model elements. In moderately simple cases, these could be created on the fly by a modeler or even a non-modeler participant, to meet *ad hoc* needs. This would only require modeling tools to offer a way to specify a simply metamodel-based query, e.g. by clicking in a tree of metamodel elements. The result of the query would be shown in a new UX as a table, tree or text (with live links to edit model elements where possible). Being able to update results automatically or on demand seems useful.

In most cases, it seems that in the resulting UX, having operations like add or delete propagated back to the wider models would present significant difficulties. In addition to implementation questions, many such cases may be semantically untenable or undesirable. The use case for non-modeler participants needing such operations is however doubtful. Indeed, at the point where operations have significant effects on the broader models, the non-modeler would become a modeler, with the attendant responsibilities of that role.

The proposed low-effort custom explorer approach seems not to have been described or implemented before. The closest cases seem to be generic *ad hoc* browsers, where a user can progress from one object to another by any possible kind of relationship. These were found in the CASE tools of the 1990s: the 'spider diagrams' of Excelerator and the automatically laid out graphical trees of TDE Navigator [15], which had the problem of being confusing: each step down in the tree from each object could be a different kind of relationship from the others, and adding new steps changed the tree layout. Using standard widgets and having consistent semantics for steps, as proposed here, may well help, as should providing ready-made queries rather than forcing each user to add the right query steps in series each time.

Clearly, other types of UX could also be beneficial and should be explored. The questions of bidirectionality of operations are also worthy of further research, both for practical solutions and for a theoretical understanding of how different language workbenches approach the question of an underlying model, its representations, their persistence and interrelatedness.

As a first step, though, it seems most productive to flesh out this vision paper's line of thought by building prototype functionality for defining a new explorer-like UX, and obtain practical experience of its use with non-modeler participants.

REFERENCES

[1] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-specific modeling: enabling full code generation.* John Wiley & Sons, Hoboken, NJ.

[2] John Hutchinson, Jon Whittle, Mark Rouncefield and Steinar Kristoffersen. 2011. Empirical assessment of MDE in industry. In *ICSE'11, May 21-28, 2011, Waikiki, Honolulu, HI, USA*, ACM.

[3] Juha-Pekka Tolvanen and Steven Kelly. 2010. Integrating models with domain-specific modeling languages. In *SPLASH Workshop on Domain-Specific Modeling 2010, Reno, Nevada.* ACM.

[4] Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax language workbench. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* ACM.

[5] Paul Klint, Tijs van der Storm and Jurgen Vinju. 2009. RASCAL: a domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation.* IEEE.

[6] Steven Kelly, Kalle Lyytinen and Matti Rossi. 1996. MetaEdit+: A fully configurable multi-user and multi-tool CASE and CAME environment. In *Advanced Information Systems Engineering: 8th International Conference, CAiSE'96* Heraklion, Crete, Greece, May 20–24, 1996 Proceedings 8. Springer, Berlin, Heidelberg, 1–21.

[7] Steve Cook, Gareth Jones, Stuart Kent and Alan Cameron Wills. 2007. *Domain-specific development with Visual Studio DSL Tools.* Addison-Wesley Professional.

[8] Charles Simonyi, Magnus Christerson and Shane Clifford. 2006. Intentional software. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* 451-464. ACM.

[9] Riccardo Solmi. 2005. *Whole platform.* PhD thesis, University of Bologna.

[10] Mert Ozkaya and Deniz Akdur. 2021. What do practitioners expect from the meta-modeling tools? A survey. *Journal of Computer Languages, vol 63, April 2021.* Elsevier.

[11] Steven Kelly. 1994. A matrix editor for a metaCASE environment. *Information and Software Technology 36, no.6.* Elsevier, 361–371.

[12] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe and Danny Weyns. 2019. Blended modelling – what, why and how. In *Workshop on Multi-Paradigm Modelling for Cyber-Physical Systems, Models 2019, Munich, Germany.* ACM.

[13] Steven Kelly and Risto Pohjonen. 2013. Dynamic symbol templates and ports in MetaEdit+. In *DSM'13 Workshop on Domain-Specific Modeling, SPLASH'13, Indianapolis, Indiana.* ACM.

[14] Steven Kelly and Risto Pohjonen. 2009. Worst practices for domain-specific modeling. *IEEE Software, July/August 2009.* IEEE.

[15] Antero Taivalsaari. 1997. Multidimensional browsing. In *Proceedings 8th Conference on Software Engineering Environments*, Cottbus, Germany. IEEE.