Automated Annotations in Domain-Specific Models: Analysis of 23 Cases

Steven Kelly[1], Juha-Pekka Tolvanen[1]

[1]*MetaCase, Ylistönmäentie 31, FI-40500 Jyväskylä, Finland*

Abstract

Modeling languages focus on expressing aspects of the system they consider relevant. These can be for instance structural, behavioral, interaction, timing etc. aspects of the system. Most languages, however, do not explicitly visualize aspects related to correctness, consistency or incompleteness of models, or visualize information from system execution or animate the models. Such domain-specific information, however, is often considered when building domain-specific languages — after all, these languages want to follow the constraints and rules of a particular domain. We examine how domain-specific modeling automates the display of such information in models through annotations. We form a categorization of the semantics and visual representation of such information, based on earlier research and our own experience. The categorization is then applied to review the use of annotations in a set of Domain-Specific Modeling languages. The study shows a wide range of annotations being displayed in both graphical models and separate text views. Based on the languages that have been used the most, we conclude that annotation increases and becomes more graphical, particularly to help new users.

Keywords

Domain-Specific Modeling, Visualization, domain knowledge, language design

1. Introduction

Modeling languages focus on expressing and visualizing aspects of the system they consider relevant. These can be for instance structural, behavioral, interaction, timing etc. aspects of the system. Modeling languages typically follow a pen and paper mentality [1] by visualizing only the data that modelers have created. They usually do not indicate or visualize for example errors related to correctness or consistency of models, show incomplete parts, provide guidance to finalize the model, show run-time information like variable values, workload etc., or animate models based on their execution. For example, modeling languages provided by standardization organizations, like SDL or TDL by ETSI[1], UML or BPMN by OMG[2] or Archimate by OpenGroup[3] focus on the separation of concerns to be visualized (e.g. SDL or UML diagram types), enable user defined views (e.g. ArchiMate), or even alternative visualizations of the same content (e.g. communication diagram and sequence diagram of UML), but very little on visualizing other

FPVM 2021: 1st International Workshop on Foundations and Practice of Visual Modeling, Bergen, 21–25 June, 2021
stevek@metacase.com (S. Kelly); jpt@metacase.com (J. Tolvanen)
https://metacase.com/stevek.html (S. Kelly); https://metacase.com/jpt.html (J. Tolvanen)
0000-0003-0931-157X (S. Kelly); 0000-0002-6409-5972 (J. Tolvanen)

[1]www.etsi.org
[2]www.omg.org
[3]www.opengroup.org

aspects than those which the modelers have entered. If such information is needed, users must run separate checks and create their own practices and add-ons (e.g. [2]).

This is somewhat understandable if a modeling language is intended only for sketching, where correctness, consistency or completeness of models are less important. Where models are used to produce code, configuration, tests or any other artefacts, however, providing such annotations becomes far more valuable and even mandatory. We investigate how modeling languages that are used for something other than sketching recognize and visualize aspects other than what modelers have entered in models. We focus on domain-specific languages and models as these have an emphasis on expressing the domain knowledge and related rules, of which there can be many and which will vary depending on the domain targeted. Such domain-specific information is often considered when building domain-specific languages — after all, these languages want to follow the constraints and rules of the particular domain.

We analyze 23 domain-specific modeling languages made with MetaEdit+, looking at the mechanisms they provide to visualize and report errors, warnings, guidance, results from analysis, and animation of run-time execution. The study shows a wide range of annotations being displayed in both graphical models and separate text views. Based on the languages that have been used the most, we conclude that annotation increases and becomes more graphical, particularly to help new users.

We begin by discussing related research on visualizing errors and similar annotations as a way to improve quality. From that research and our own experience we create a categorization of the semantics of such checks and the ways the results are displayed to modelers. We apply the categorization to the cases, analyse the results, discuss and present our conclusions.

2. Related research

One of the foundations of Domain-Specific Modeling is that an appropriate language reduces errors. In the words of Alfred North Whitehead, "By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems." Numerous results from experiments and industrial use confirm this for Domain-Specific Modeling [3, 4, 5], both explicit and implicit (e.g. [6]). The links between graphical notations and underlying cognitive factors have been detailed well in [7] and [8].

In this paper we want to go beyond between this known effect of the language and notation, and focus on annotation: additional indications of errors and similar information about the model. This topic has been studied significantly less, and indeed is almost invisible in research: models in publications are unsurprisingly normally shown without errors.

The earliest research stems from the CASE tools of the 1980s, with [9] describing trade-offs familiar to DSM implementers today: error avoidance vs. error highlighting, unobtrusiveness vs. detailed messages, coverage vs. performance etc. Their research highlighted the difficulties of applying the familiar compiler or IDE lists of errors, based on a sequential, line-based textual input, to the richer world of hierarchical graphical notations. They favored displaying graphical annotations as coloring of existing notation elements or text. Errors in model information that is not in the surface notation could be displayed by a separate command to list those elements: selecting an element would display that portion of the model.

Rech and Spriestersbach [10] investigated model defect annotations, with a focus on UML. Their survey found that 90% of users want such annotations, which they divided into several methods of representation. A separate view containing an IDE-like list of errors received the highest ratings, particularly for familiarity, preciseness, consistency and understandability. Its weaker points were ease of use, appealingness, and usefulness, where graphical annotations came close or even beat it. Graphical annotations seemed to divide into those based on adding an icon, which fared a little better than those that added an error color to existing symbol or text elements in the notation. Coloring was clearly preferred to bold formatting, and there was little difference between whether the coloring was applied to text, underlining, or as symbol element line, fill or glow color.

In more general computer user interfaces, by far the most studied area for error annotation is web forms. As these brought an order of magnitude increase in the number of people performing data entry, this is perhaps unsurprising. Research by pioneers like Jakob Nielsen [11] before this growth has thus been supplemented based on experiences with less technical users — in a way that may be analogous to the move from programmers to domain experts in Domain-Specific Modeling. This new research often has clear financial benefits, making adoption a matter of fact rather than taste. Baymard Institute benchmarks [12] revealed that live inline validation of form input grew from 13% in 2012 to 60% in 2016. Performing validation at the right time was key: showing every keypress in a social security number field as an error until the full number was entered was too early and distracting; only showing errors after submission of the whole form was too late and disorientating.

While an error in a model may have significant effects, an unnoticed error condition in an industrial system may be catastrophic. TEPCO, the operators of the Fukushima nuclear plant, have continually revised and improved their SAFER (Systematic Approach For Error Reduction) method [13], whose first four steps map well to the approach of DSM:

1. Stop/Eliminate -> make that work no longer necessary: raise level of abstraction
2. Cannot be performed -> prevent dangerous action altogether: language rules
3. Make easily understood -> language is domain-specific
4. Make easy to perform -> effective tooling

The SAFER method uses the earlier m-SHEL model [14], which includes as a strong element the frailty of the 'Liveware' element — the human using the system — but with the focus on the failure of the system designer to take these human elements into account, rather than humans being 'faulty' for being human. Although annotations could be added for any number of errors and warnings, at some point they being to compete for attention with actual model content. This distracts the users from focusing on the facts in the elements they are working on, drawing the eye and concentration away onto annotations outside that area. In a nuclear power station, drawing the attention away from current work may be merited: the indication is often time-sensitive. In a model this is rarely necessary: the abstract thinking needed to make a good design often approaches the cognitive limit of the modeler anyway.

3. Categorizing annotations

We include as annotations all displays of information that come by analyzing the model data and presenting added value to the modeler. These go beyond the simple display of model data that form the notation of the language, and the generation of code or other text that form the desired output. We categorize them below based on their semantic content (error, warning etc.) and their visual representation in the tool (icon, textual report etc.).

3.1. Semantic content of annotations

For identifying and analyzing visualizations, we derived a category based on practices that can be found from development tools (e.g. programming tools, modeling tools), namely:

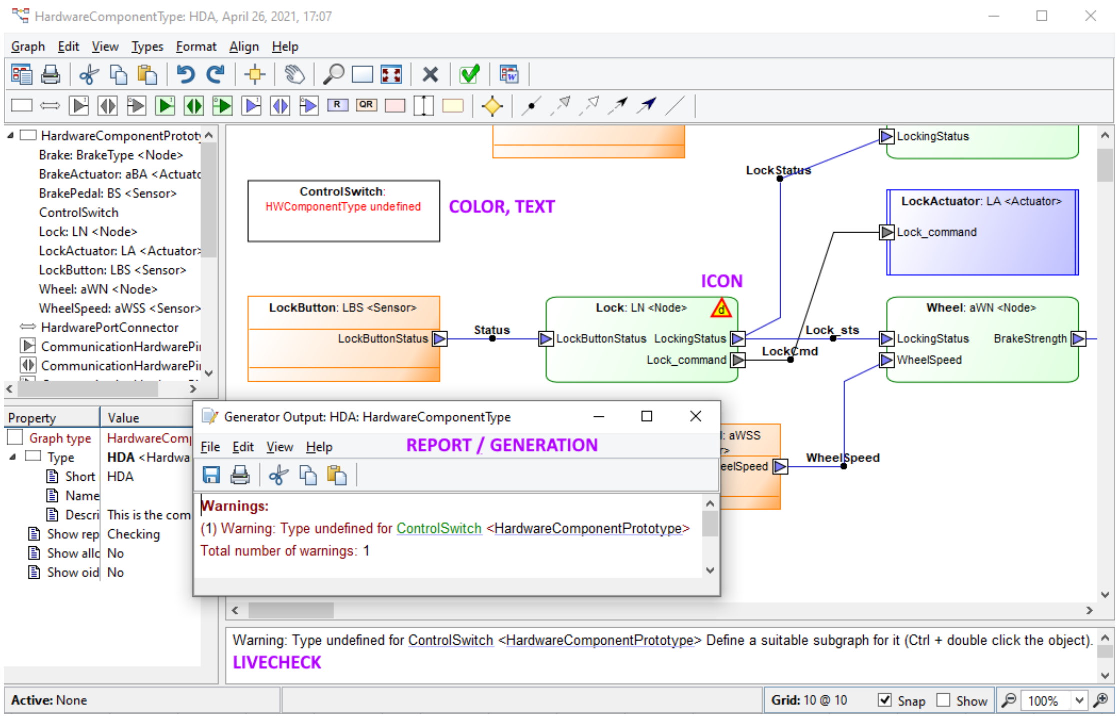
- **Error**: the model does not generate or compile. For example, relevant data is missing or the model has inconsistencies.
- **Warning**: the model generates or compiles but has some possibly incorrect or unfinished data, e.g. elements that are unconnected or unused.
- **Guidance**: suggestions to the modeler of what to do next (other than correcting errors or warnings). For example, adding extra model elements to further construct the model.
- **Results of analysis**: show results of static or dynamic analysis of the model, e.g. calculated values, results of coverage or performance tests.
- **Animation**: the model is animated based on its execution, e.g. by highlighting the currently executed part or showing real-time values.

It is worth emphasizing that since we focus on domain-specific languages, many domain rules are already covered by the language definitions themselves. In other words, the language already prevents the creation of illegal or unwanted models. Similarly, the tool may read the language definition and stop some actions based on existing model data, e.g. preventing a duplicate name. Together, these decrease the need for showing errors or warnings, and probably reduce occurrences of those categories compared to general-purpose languages and tools.

3.2. Visual representation of annotations

Rech and Spriestersbach [10] evaluated methods of visual representation in tools (icon, color, bold, dashed, aura, form, size, pattern, opaque, tilting, views) and settled on six for their survey: icon, color, bold, underscore, aura, views. As they did, we exclude those that clash with notation (form, size, pattern, tilting), and also the poorest from their survey results (bold). Several of their categories were rather close and received similar scores in their survey (color, underscore, aura), so we collected these into one category, color. In our experience, a common annotation method is the addition of text, with or without color, to a symbol, so we added that. This gave us three categories of annotation that form part of the graphical model:

- **Icon**: an extra graphical element displayed as part of a model symbol
- **Color**: enhancing the visibility of a model symbol element through color
- **Text**: a new text added to a model symbol



Warnings:
(1) Warning: Type undefined for ControlSwitch <HardwareComponentPrototype>
Total number of warnings: 1

Warning: Type undefined for ControlSwitch <HardwareComponentPrototype> Define a suitable subgraph for it (Ctrl + double click the object).

Figure 1: Visual representations of annotations.

The concept of views — textual outputs listing errors etc. in a separate pane or window — was found to be rather broad, so we divided it into three in line with our experience. These followed the facilities offered in MetaEdit+:

- **LiveCheck**: run after every operation and displayed in a pane below the model
- **Report**: run on demand and displayed in a separate window
- **Generation**: run and displayed when the user generates code or other desired output

Figure 1 illustrates these 6 visualizations (labeled in purple).

4. Research method

To examine how Domain-Specific Modeling languages apply annotations we chose an empirical approach. To complement the study of Rech and Spriestersbach [10], that surveyed modelers about their preferences in hypothetical cases, we looked at real-world models as concrete data, to see what has actually been used — and evolved through extended use. We selected a broad set of DSM solutions from practical cases, and analyzed them to identify the annotation approaches they used.

4.1. Case selection and analysis

The data for the analysis was gathered from cases of DSM in practice, starting with an initial set of over 100 candidate cases, and narrowing it down to 23 that suited this topic well yet formed a broad cross-section. We sought breadth in the range of problem domains, from insurance products through aerospace systems to AI bots. The cases were also chosen to cover different kinds of modeling needs and use of generated outputs, such as DSM solutions whose main functionality is to produce fully functional executable code, tests, configuration, simulation, analysis or documentation in a specific format. Finally, we wanted cases from a broad range of history, from 1995 to 2021, of amount of use, from days to decades, and of users, from one to hundreds. Although it was not a criterion, we also ended up with a reasonable mix of cases from three earlier papers [15, 16, 17] and previously unreported cases.

An important criterion for the cases was that access to language definitions would be available to allow full analysis of the language definitions, as well as to see various annotation approaches in practice. All modeling languages and generators were implemented in MetaEdit+ [18]. The categorizations of annotation semantics and visualization, however, are not limited to any particular tool.

The categorizations for the cases were performed by both authors, each checking the other's results to ensure that categories were applied consistently. Analysis of the resulting data was performed mainly by the first author, benefiting from shared discussions.

5. Analysis

Table 1 shows the cases' problem domains, brief information on history and use (columns Note – Users), and the annotation approaches found, according to their visual category (columns Icon – Generation) and semantic category (letter within those column: Error, Warning, Guidance, Results of analysis, Animation). Confidentiality prevents more detailed information on cases.

Figure 2 plots each case with its X co-ordinate being a 'usage score' based on the four history columns, multiplied to show increasing usage (see Table 1 Notes), and the Y co-ordinate being the number of annotation categories found in that case (letters on that row in the table).

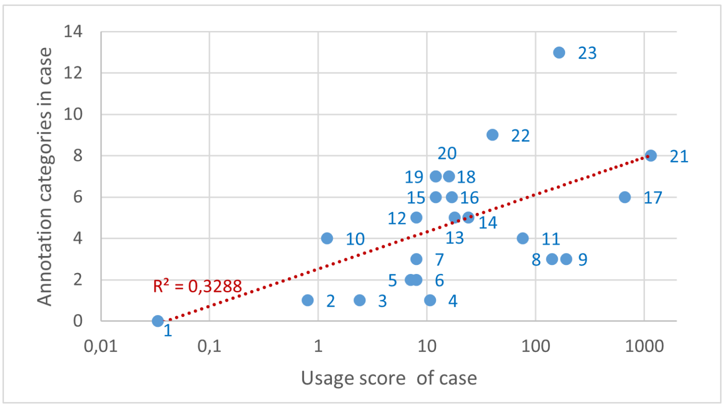


Figure 2: Correlation of annotations & DSM language usage.

Table 1
Count of cases including each semantic and visualization category of annotation

ID	Problem domain	Note	Years	Phase	Users	Icon	Color	Text	LiveCheck	Report	Generation
1	Home automation	*	0.1	1	1						
2	Database applications		0.2	2	2			G			
3	Data architecture		0.2	2	6					E	
4	Insurance products	*	2	4	4					W	
5	Insurance systems	* 3	4	4	40					R	R
6	Enterprise applications	3	5	4	12						E W
7	Big data applications		1	2	4				E W	E	
8	Phone UI applications	* 2	8	4	400					R	E R
9	Government EA		4	3	16		R			E W	
10	AI bot		0.3	2	2	W	A	W R			
11	Call processing	*	19	2	6				E W	E W	
12	Medical		2	2	2	R		R	E	E	E
13	Security		1	3	6		R	R	E G	E	
14	Industrial automation		4	3	2			G		E W	E W
15	Consumer electronics		6	2	1				E W	E W	E W
16	Blockchain ecosystems		0.25	2	34				E W G	E W G	
17	Software testing		3	4	55			E	E W	E W G	
18	Telecom		3	2	2	W	W	E W		E W	R
19	Performance testing		3	2	2	R A	E G	E G			E
20	Aerospace		4	2	2				E	E W R	E W R
21	Consumer electronics		12	4	24		E G	E W G		E W R	
22	Automotive ECU		5	4	2	E	E	E	G	E W	E W R
23	Automotive architecture	1	11	3	5	G	E W	E W G	W G	E W R	E W

Notes: * Before graphical condition support (usage score 1/3); Pre-existing: 1 language, 2 generator, 3 both (2 & 3: usage score 1/30)
Phase: 0 sketch, 1 Proof of concept, 2 pilot, 3 use, 4 major use
Columns Icon – Generation: Error, Warning, Guidance, Results of analysis, Animation

5.1. Graphical annotations require tool support

Tool features are an obvious prerequisite for graphical display of annotations. The tool must offer the metamodeler features to obtain the necessary information from the model, and features to display the annotation. In early MetaEdit and MetaEdit+ versions, symbols from the metamodel were displayed as-is, with only the property value text varying — leaving no scope for annotations. In MetaEdit+ 3.0 in 1999, conditionality was added to symbol elements, so they could be set to display only if a given property value matched a given wildcard string. This allowed more visual variation in symbols, e.g. different arrowheads or different colored icons for public, private and protected objects. However, the conditions needed for annotations are often more complicated, and the full scope was only really covered in MetaEdit+ 4.5 in 2006, where scripts in the MERL generator language could be used as the conditions for symbol elements, and also to produce the content of text elements. Otherwise, tool version had no effect.

Of the 23 cases, 12 had essentially no graphical annotations, and of these 5 were because the languages were made in tools without the prerequisite features.

5.2. Graphical annotations are added over time as use grows

The amount of checks and guidance seems to be influenced initially by time: many metamodelers only begin adding these after passing the initial early stages of language evolution. When a language starts being used, a major factor seems to become the number of new users who have to learn the language and then independently use it. A language can be used for over a decade with virtually no checks or guidance, if there is only the small set of initial users; if a language is introduced to many new users, and particularly if their use is only for a short time or occasional, the need to provide checks and guidance is greater.

When ordered by increasing number of annotation categories used, as in Table 1, the growth in category use proceeds rather clearly from the lower-level textual views on the right to the higher-level graphical views on the left. This follows our own intuition in the ordering of the categories, and also the results of Figure 9 in [10]: a conditional symbol icon is preferred, followed by colored text, then LiveCheck is preferred over Check, but all are seen as useful.

Of 12 early-stage languages, only 3 had substantial use of graphical annotations; 7 had no graphical annotations and 2 had only a single graphical annotation.

Of the 11 languages with significant use (and excluding 3 where tool support was not available), all but 1 had substantial use of graphical annotations.

All languages with significant use included text view annotations. Even among early-stage languages, only 3 at the earliest stages had no text view annotations, and only one of these — the only one at the proof-of-concept stage — had no annotations at all.

Of 7 languages with only one or two users, 2 had no graphical annotations and 3 no text view annotations.

Of 4 languages with over 20 users, all had both graphical annotations (where tool support was available) and text view annotations.

Although the languages with most annotation categories were from automotive and aerospace domains, and database and insurance domains had the fewest, the former were also among the longest projects. More data would be needed to separate out the effect of the domain.

Table 2
Count of cases including each semantic and visualization category of annotation

	Error	Warning	Guidance	Results of analysis	Animation	Total
`Icon`	2	2	1	2	1	8
`Color`	5	1	3	2	1	12
`Text`	5	4	5	2	0	16
`Graphical`	12	7	9	6	2	36
`LiveCheck`	8	6	5	0		19
`Report`	14	11	1	4		30
`Generation`	8	6	0	4		18
`Text views`	30	23	6	8		67
`Total`	42	30	15	14	2	103

5.3. Counts of cases including annotation categories

Table 2 shows that nearly twice as many annotation cases were in text views as in graphical annotations. In graphical annotations, the addition of a text element (with or without coloring) was more common that color emphasis, and icons were the least common method. Although an icon is visually concise, it is also severely limited in the information it can present – generally requiring further elaboration e.g. in a LiveCheck pane, which requires extra cognitive work by the user to link the information. Presenting the same text (or a concise form of it) in the symbol itself is often a better choice.

Providing annotation results only during generation was the least common method of using text views. In our experience, users prefer to be able to invoke the annotation check in a separate report. Where performance allows and the results are not too distracting, display in a LiveCheck pane is also useful.

The most common semantic content of annotations was an error, followed by warning, guidance, results of analysis and animation in that order.

6. Discussion

Adding annotations in various categories is not a big effort for language engineers. From industrial cases we see that creating a whole DSM solution takes on average 10 days [19], and including annotations is a small fraction of that. Tools may influence the effort required, e.g. if traditional programming is needed to implement editors, or symbols must be made manually in XML. A study by El Kouhen et al. [20] indicates that some tools may require 50 times more effort than others.

The availability of annotations was seen to correlate positively with longer usage by more users. Causality may well be active in both directions: annotations improve acceptance, and more users increases the pressure for annotations and also the payback on defining them. The return on investment is also improved in the common case that the same annotation logic can be applied in several visualizations, as seen in Figure 1, where four different annotations all note the same undefined type.

A limiting factor for this research was the availability of sufficient suitable cases. Although being able to use industrial cases makes the results more representative, confidentiality agreements prevent full disclosure of the languages. Having all the languages in one tool improved our consistency and allowed some automation of the analysis. However, there is the possibility that some findings would be different if applied in another tool, with different costs of implementing annotations, lacking some annotation categories, or offering others. This threat to validity is ameliorated by the fact that the results here, in [9] and [10] largely support each other.

7. Conclusions

Although Domain-Specific Modeling languages rule out a significant portion of errors found in generic modeling or programming languages, automated annotations to show errors and other analyses of models seem to be useful to modelers. Increased use correlates with an increase in annotations, and in particular with more graphical, real-time annotations.

These results of the analysis should hopefully be useful to practitioners, allowing them to compare their language with others and offering hints of possible annotations that others have found useful. The clear increase of annotations with language usage is an indication to researchers and tool builders of the need for such facilities, particularly as DSM use scales up.

More research on the subject is still needed. Extra factors could be included, such as the scope of the model that is analyzed for an annotation: a single object, a diagram, a diagram and its sub-diagrams, or the whole set of diagrams. A larger set of DSM solutions could be analyzed, and coverage could be extended to tools other than the one used in this study. This could also widen the categorization of annotations: we did not include as annotations the information that the user can already obtain in any language with the generic features of MetaEdit+, such as places an object is reused, nor model-to-model or layout transformations such as in [21].

References

[1] J.-P. Tolvanen, K. Lyytinen, Flexible method adaptation in CASE, Scandinavian Journal of Information Systems 5 (1993) 51–77.

[2] A. Mattsson, B. Fitzgerald, B. Lundell, B. Lings, An approach for modeling architectural design rules in UML and its application to embedded software, ACM Transactions on Software Engineering and Methodology (TOSEM) 21 (2012) 1–29.

[3] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, L. Walton, A software engineering experiment in software component generation, in: Proceedings of IEEE 18th International Conference on Software Engineering, IEEE, 1996, pp. 542–552.

[4] S. Kelly, J.-P. Tolvanen, Domain-specific modeling: enabling full code generation, John Wiley & Sons, 2008.

[5] J. Kärnä, J.-P. Tolvanen, S. Kelly, Evaluating the use of domain-specific modeling in practice, in: Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling, Helsinki School of Economics, 2009, pp. 14–20.

[6] Q. Cao, F. Nah, K. Siau, A meta-analysis on relationship modeling accuracy: Comparing relational and semantic models, AMCIS 2000 Proceedings (2000) 14.

[7] T. R. Green, Cognitive dimensions of notations, People and computers V (1989) 443–460.

[8] D. Moody, The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering, IEEE Transactions on software engineering 35 (2009) 756–779.

[9] R. F. Gordon Ph D, P. G. Loewner, E. A. MacNair, K. J. Gordon, J. F. Kurose, H. Wang, Error detection and display for graphical modeling environments, in: Proceedings of the 1991 Winter Simulation Conference, 1991, pp. 1139–1145.

[10] J. Rech, A. Spriestersbach, A survey about the intent to use visual defect annotations for software models, in: European Conference on Model Driven Architecture-Foundations and Applications, Springer, 2008, pp. 406–418.

[11] J. Nielsen, Enhancing the explanatory power of usability heuristics, in: Proceedings of the SIGCHI conference on Human Factors in Computing Systems, 1994, pp. 152–158.

[12] Baymard Institute, Usability testing of inline form validation, 2016. URL: https://baymard.com/blog/inline-form-validation.

[13] TEPCO, Disaster analysis method — SAFER, 2021. URL: https://www.tepco.co.jp/en/hd/about/rd/safer-e.html.

[14] F. H. Hawkins, Human factors in flight, Gower Technical Press, 1987.

[15] J. Luoma, S. Kelly, J.-P. Tolvanen, Defining domain-specific modeling languages: Collected experiences, in: 4th OOPSLA Workshop on Domain-Specific Modeling, University of Jyväskylä, 2004, pp. 1–10.

[16] S. Kelly, R. Pohjonen, Worst practices for domain-specific modeling, IEEE software 26 (2009) 22–29.

[17] J.-P. Tolvanen, S. Kelly, How domain-specific modeling languages address variability in product line development: Investigation of 23 cases, in: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A, 2019, pp. 155–163.

[18] S. Kelly, K. Lyytinen, M. Rossi, MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment, in: International Conference on Advanced Information Systems Engineering, Springer, 1996, pp. 1–21.

[19] J.-P. Tolvanen, S. Kelly, Effort used to create domain-specific modeling languages, in: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, 2018, pp. 235–244.

[20] A. El Kouhen, C. Dumoulin, S. Gérard, P. Boulet, Evaluation of Modeling Tools Adaptation, Technical Report, CEA, 2012. URL: https://hal.archives-ouvertes.fr/hal-00706701v2.

[21] C. F. Lange, M. A. Wijns, M. R. Chaudron, A visualization framework for task-oriented modeling using UML, in: 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07), 2007, pp. 289a–289a. doi:10.1109/HICSS.2007.44.