Empirical Comparison of Language Workbenches

Steven Kelly
MetaCase
Ylistönmäentie 31
40500 Jyväskylä, Finland
+358 14 641000 ext. 21
stevek@metacase.com

ABSTRACT

Production use of Domain-Specific Modeling languages has consistently shown productivity increase by a factor of 5–10. However, the spread of DSM has been slowed by projects stalling even before the language was built, often citing problems with the chosen tool. With a wide variety of language workbench tools for DSM, there is a need for objective empirical tool comparison – particularly as the little research so far shows a range of 50 times more effort between the most and least efficient tools. This article looks at existing empirical research and an experimental design for a future comparison. We aim at increasing objectivity and repeatability while keeping overall effort practical, and providing worthwhile returns for the participants.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE).
D.2.6 [**Programming Environments**]: Graphical environments
D.2.8 [**Metrics**]: Product metrics, Performance Measures
G.3 [**Probability and statistics**]: Experimental design

General Terms

Measurement, Performance, Design, Economics, Reliability, Experimentation, Human Factors, Languages.

Keywords

experiment design, language workbench, comparison, survey, quantitative, qualitative.

1. INTRODUCTION

Where Domain-Specific Modeling languages have made it into production use, the results have been promising: consistently high increases in productivity [1]. However, many projects that have looked at using DSM have stalled before production or even piloting. Aside from usual reasons that affect all projects, one of the most common complaints has been about the chosen tool – the language workbench or its resulting modelling tool. With a wide variety of language workbenches for DSM, there has been a surprisingly small amount of empirical research comparing them.

The evidence seems to be that there is a large difference in how much work is needed to achieve the same tool support for a DSM language with different language workbenches: some workbenches require 50 times more work than others [2]. Language developers who are also programmers may prefer the freedom of working with a framework rather than a tool, but that can be 2000 times slower [3]. Even among mature tools specifically designed for DSM, research can show an order of magnitude difference between the top two [2] (Figure 1 below).

This article aims to form a preliminary investigation of the area of empirical comparison of language workbenches. We will look at the particularly challenges of quantitative comparison in this area, the different factors that could be compared, and previous comparisons. Based on this we will offer a suggestion for an experimental design appropriate for a future comparison.

2. CHALLENGES OF COMPARISON

Conducting an empirical comparison of language workbenches is difficult, particularly given the wide range of effort required for the same results. In most cases, only an unrealistically small language could be built purely for an experiment: otherwise using the less efficient tools will take too long. The more realistic data available from full-sized industrial cases will always – by the definition of domain-specific – be based on building different languages, and hence be unable to help. Even if the same team builds the same language again with another language workbench, learning effects will undermine the comparison: not only is the language and task more familiar, but most likely the earlier case was their first DSM project anyway.

Empirical research in industry has thus tended to concentrate on comparing the productivity of building systems with the DSM languages to the pre-DSM productivity, and setting that against the effort to create the DSM solution. Those figures have direct value to the company in assessing which approach to use to build their products, and also in estimating the up-front cost and return on investment of using DSM on another domain. Another company looking at the results can see that the tool used was successful (or not), and hence whether it is worthy of their consideration, but not whether it is better or worse than another tool would have been.

Several different kinds of comparison are thus possible, e.g.:

a) comparing language workbenches as different ways of producing a DSM tool for the same language;
b) comparing the effort to update the resulting DSM tool when the LWB, problem or solution domain evolves;
c) comparing the productivity of the resulting DSM tool and generation against hand-writing the same code;
d) comparing the productivity of different languages made for the same domain with different workbenches;
e) comparing the performance of the resulting DSM tool: how long the user has to wait for the tool to open a model, generate code, show model changes etc.

All of these are interesting and useful comparisons, but for reasons of space and focus we will concentrate here on a). Although b), c) and d) have a higher economic influence, they only become relevant once a) has been accomplished, and that is the main hurdle facing DSM today.

3. BASIS OF COMPARISON

Even when we have decided what to compare, choosing a good basis of comparison is by no means an easy task. Using a modeling tool requires a certain level of ability; creating a modeling tool with a language workbench requires more, often being left to the top developer in an organization. Even more is required of a language workbench creator, so attempting to rise still higher to compare many language workbenches will be a humbling experience for anyone.

Fortunately, we are not working in a vacuum, nor as the first to attempt such a comparison. By looking at how comparisons are performed in more mature areas, we can establish some ground rules. And by looking at previous language workbench comparisons, we can see the practical challenges and explanatory strengths and weaknesses of various bases of comparison. Such bases include feature coverage, lines of code, user satisfaction, time, and cost. Qualitative approaches such as questionnaires provide vital extra information to interpret and apply this data, but for reasons of space we focus here on quantitative metrics.

3.1 Feature coverage

A large portion of the literature on language workbenches is composed of descriptions of the authors' own new or improved language workbench. Academic rigor demands a 'Related Work' section, but other factors tend to push such a description into focusing on features where the new workbench excels. The quality of investigation into others' language workbenches is unencouragingly low. The majority of authors apparently do not even trouble to download the other tools, and many statements are so clearly wrong that the first search engine result on the topic would have corrected them.

In addition, the interpretation of a feature defined in a couple of words is highly subjective. Attempts to add more explanation tend only to narrow the definition down to fit only that implementation of the feature present in the author's own workbench. Even with a common understanding of a feature, there is generally no clear standard or agreement on what level of support must be present to say the feature is present.

Most seriously, however, the presence or absence of most features is not yet proven to have any effect on actual performance or usefulness of the language workbench. Indeed, the things that neophyte users expect, or a new workbench creator may decide to implement, surprisingly often turn out to be a false step in terms of the end results – a feature found in many tools may turn out to be the GOTO of the workbench world.

3.2 Lines of code

As with programming languages, the number of lines of code required to implement a system is a tempting basis for comparison. However, even with improvements to filter out the effects of white space and comments, research shows that lines of code is a poor comparitor of development effort between two projects – even by the same team in the same language. A line of code for some complicated issue may take orders of magnitude longer than for a simple issue. Measuring only the final lines of code also ignores that a given line of code may have been rewritten many times or even deleted.

Where those projects are performed by different teams, and in particular with different languages, the comparison breaks down completely. In this case, the system would be the whole DSM solution – the language concepts, rules, concrete syntax, semantics (by generation or interpretation), and editor tooling. The construction of these different areas requires different languages even within one workbench, meaning that we are comparing apples + oranges against pears + grapes. As some of the areas are not even created with textual languages, but with a graphical language or by using a user interface, we also face the task of trying to map those approaches into some kind of textual syntax, possibly created just for the experiment. It goes without saying that the reliability of such results approaches zero. An order of magnitude difference may still be visible, but a tool could easily be twice as good as another yet obtain worse results.

3.3 User satisfaction

In many ways, user satisfaction is the most honest and useful measure of tool success. If we want to predict which programming language or tool will be chosen for the next project, knowing which the users like the most will give us more information than knowing the feature list or how many lines they will have to write. In a non-commercial setting, and if decisions are made by the users from their own point of view, their favorite tool will be the winner almost every time.

In a commercial setting, or where other factors encourage a broader and more long term view, user satisfaction becomes just one part of the equation. The desire for familiarity and apprehension about moving outside one's comfort zone may be outweighed by a sufficiently large and well-attested benefit of using something else. Measuring of user satisfaction before and after such a change will be a useful exercise. However, comparing several language workbenches in this way is limited by the longitudinal nature of such comparisons. In addition, user satisfaction is not only highly subjective, but significantly affected by things like team spirit and general mood, unrelated to the tools in question.

3.4 Time

Like lines of code, time is something that is objectively measurable. It is most reliable when used to measure tasks comfortably within a user's ability, but may tend to infinity for tasks near the limit of that ability. The measurement is easy and largely repeatable for experiments, but harder for industrial use, where users may work on many other things throughout the day. Separating out work on different areas is easier where the task is more focused on tool use rather than extended periods of thinking.

Unlike lines of code, time is equally comparable across languages, whether in one or several workbenches: we are adding hours + hours, not apples + oranges. It also reveals the effort spent on multiple versions of the same line, or on lines that were eventually deleted. In academic situations, and in commercial situations where time-to-market is critical, time is itself the variable of interest; in non-time-critical commercial situations, it is an excellent proxy for the primary variable of cost.

Whether time spent or lines of code produced are a better measure of the size of a project, in terms of possible future maintenance effort, is unclear. Time does however reflect the cost of learning better than lines of code; whether that is desirable depends on the experiment – whether we want to know how long a tool takes to learn, how long the first project will take including learning time, or how long projects will take going forward.

3.5 Cost

For many commercial situations, and some academic situations, the overall financial cost of creating and providing users with a DSM tool, and having them use it to produce systems, is the primary variable. This includes the initial and maintenance costs of the tool, and the time of the language creator and users. This cost is then compared against alternative ways of producing the same systems.

The normal trade-offs between commercial and free tools apply, but here there are two levels: firstly that of using the language workbench to produce a DSM tool, and then using the resulting DSM tool to produce systems. In all but the smallest cases or most inefficient language workbenches, the overwhelming factor is the productivity of using the resulting DSM tool. Assume a modeler produces as much as he costs, at a fully-weighted cost per month of 4,000€. If the DSM tool increases his productivity by a factor of 7.5, his value per month is 30,000€. Compared with this increased value of 26,000€, even the most expensive tools cost less than 1% — less than the difference between a productivity increase of 7.5 or 7.6.

As the challenge of DSM introduction is more that a project stalls at the language development stage because of problems with the tool, rather than because of a difference in final productivity of 7.5 rather than 7.6, cost seems not to be a particularly revealing variable. Trying to obtain a generally and globally applicable total cost comparison is difficult because of marked differences in developer salaries, different tool prices in different territories, volume discounts, and non-public pricing. While cost is thus not objectively applicable in such a comparison for general use, it can easily be factored in by a particular company into their own calculations based on language development time and productivity.

4. PREVIOUS COMPARISONS

In 1993, in what is probably the first language workbench comparison, Marttiin et al. [4] compared three tools (QuickSpec, RAMATIC, and Customizer), offering a framework for comparison that took into account the different tasks in language development and the effectiveness of the tools for carrying out the task. They used the five languages of the SMARTIE method as sample languages to be implemented in all the tools. At this early stage of language workbenches, the focus was primarily on whether the tools could faithfully implement the various features of the languages. A positive feature of the comparison, missing from many, was that the authors contacted the workbench makers for support and to ensure the reliability of their results.

Isazadeh [6] compared five graphical language workbenches (Metaview, Toolbuilder, MetaEdit+, 4thought and CASEmaker) in terms of features, architecture, and ability to model the same sample language, a variant of finite state machines. The results of the empirical experiment were subjective ratings for how simple each tool made six areas of work such as concepts or complex constraints. Toolbuilder and MetaEdit+ came out equal top, with tools in general splitting into those like the former that offered a "very high level of expressive power" but made all tasks "very difficult", and those like the latter that focused on making common tasks easy. This is the normal trade-off between low-level and high-level approaches: neither approach is the correct one, the contingencies of a situation determine which to choose.

After these early comparisons, we will not consider the large number of non-empirical or purely feature-based comparisons, focusing instead on those that provide quantitative comparisons.

Kelly and Rossi [5] performed a laboratory experiment to compare graphical and matrix-based metamodeling in MetaEdit+. 11 students were trained in both, randomly divided into the two groups, and given 3 hours to metamodel parts of the nascent UML. Their results were graded on the accuracy and completeness of various categories of metamodel elements, e.g. entities, unary relationships, and binary relationships. The hypothesis that matrices would help on relationships was supported (73% score for matrix users, 54% for diagrams); elsewhere there was no clear difference. To our knowledge this is the only test of the contribution of a particular feature of a language workbench to metamodeler performance – hopefully we will see more in the future.

Kelly [3] compared an existing Eclipse GEF implementation of a graphical logic gate language with the time required to model a similar language in MetaEdit+. A COCOMO estimate was used to transform the GEF Java code size (332KB, 120 files, over 10,000 lines) into a time value of 13 man-months (2000 hours), which compared unfavorably with the one hour required to create the same results in MetaEdit+. Using standard programmer LOC productivity figures to convert Java lines of code into time allowed a comparison across different tools and languages, where the latter tool also had no textual syntax. Using an existing implementation in the slower tool allowed a comparison to be performed within a reasonable time, despite the wide range of speeds; copying another tool's implementation rather than a neutral specification puts the latter tool at a slight disadvantage, which may narrow the speed difference found.

Pelechano et al. [8] compared Microsoft DSL tools with Eclipse EMF+GMF+MOFScript, having roughly half of 48 students use each tool. The students used the tools in a 1-semester laboratory course to build a DSL and code generation, and answered a questionnaire after the course. Most questions were a mix of feature support and user satisfaction. Each student chose their own DSL and generation target, most of which seem to have been code generation from UML diagrams; there does seem to have been metamodeling in all cases. Students did not get a chance to try the other tool, but when asked at the end whether they would use the same tool or another, 100% of the Eclipse users said they would remain faithful. The other answers also indicate a preference for Eclipse; it is not revealed whether Eclipse and Java, or Visual Studio and C#, were equally known by the students before. As the former are far more common in universities, preference for the familiar would seem to be a threat to validity in this and other studies using students. A positive factor, often missing from comparisons, is that the exact versions of the tools used are stated.

A bias to familiarity with Eclipse seems to have been avoided in Özgür's Master's thesis [9], which compared Microsoft DSL Tools and Eclipse EMF+GMF, developing the same business entity language with each. He also compared UML, MDA, Software Factories, and Domain-Specific Modeling as approaches, again seemingly more objectively than most. He found both toolsets usable, with Microsoft's being easier to use and Eclipse supporting OMG standards. Unfortunately, no quantitative figures are provided in this otherwise well-rounded thesis.

De Smedt [7] compared his department's AToM3 with MetaEdit+ and Poseidon on a simple road traffic language as a student project. He found he was slightly faster with MetaEdit+ than AToM3. The time figure for Poseidon was only a quarter of that for the other tools, because it had no functions for and was thus unable to attempt the transformation and simulation tasks. (This kind of omission or comparison of unlike work is common in student projects and Master's theses: a Spanish project [10] compared MetaEdit+ and DSL Tools, but left out concrete syntax time from the total time for DSL Tools "because it took so long", while including the corresponding time for MetaEdit+.) The known threat to impartiality, where a comparison includes a tool from the author's own organization, is exacerbated in cases where a student submits his assessment for grading by his superiors who made the tool.

El Kouhen et al. [2] produced what may be the best empirical comparison of graphical language workbenches to date. They created BPMN support in each of five tools (RSA, GME, MetaEdit+, Obeo, and Eclipse GMF). Rather than students or makers of the tools, their users were the authors: Eclipse committers on the Papyrus project, which aimed to produce its own language workbench. They graded the tools on various features, but also on the total time taken to create the abstract syntax, concrete syntax and rules (Figure 1). The wide range of results is particularly interesting given the users' familiarity with Eclipse GMF and ECore, which is also used in RSA and Obeo.

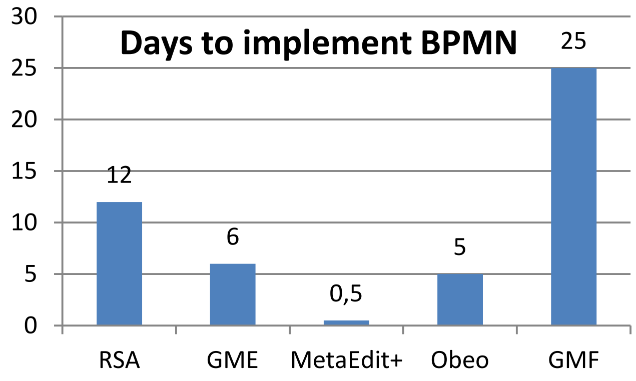

Figure 1: Days to implement BPMN [2]

A notable omission from the report is the lack of reference to related work. Had it been submitted for peer review this would no doubt have been corrected. The reason for not submitting these results as a publication, particularly after so much work, remains an open question. The report is also a good example of the problem of guessing features which would contribute to effective usability of a language workbench. The authors spend three pages listing a variety of factors that they measure from the tools before the experiment to obtain a "usability" percentage, divided into "efficiency", "task visibility" and "visual coherence". The tool with the worst "efficiency" and "task visibility" percentages turned out to be the fastest, and the tool with the best "task visibility" percentage turned out to be the slowest. Looking at the criteria, it seems they are more measures of how much the tools follow the user interface patterns familiar in Eclipse, rather than anything objectively good. Indeed the actual measured speed of use of the tools may point to a need to re-evaluate some user interface decisions that Eclipse users have become accustomed to expect. Similarly the common programmer's desire for every detail of a task to be visible appears to run counter to productivity: e.g. only by hiding unnecessary details can third generation programming languages be more productive than assembly language – a principle surely familiar to all DSM practitioners.

The report [11] from the 2013 Language Workbench Challenge is the largest comparison of recent times, including mostly textual but also graphical and projectional workbenches. All the tools implemented a subset of a questionnaire language, presented textually but amenable to other representations. Most implementations were made by tool experts, often the tool developers: a single user for most, but five for MPS and several for Spoofax. The main purpose of LWC was to present the results to each other, and development time and conditions were not measured or controlled. The post hoc idea of writing an article including measurements could thus only rely on feature coverage and lines of code, although it was known these were not well comparable. Some tools generated JavaScript (vanilla or with libraries like jQuery), others Java; Spoofax generated an existing Spoofax DSL; still others used interpretation within the workbench.

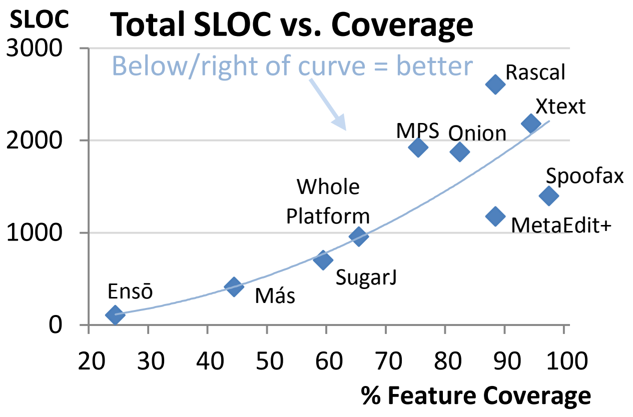

Figure 2: Lines of code to implement LWC2013 features [11]

It was found that lines of code per feature increased with feature coverage: the easier features were done first, the later ones being inherently harder or needing more lines of code as lower-level facilities were used. Figure 2 shows the data obtained, along with Excel's best fit power curve. It is interesting to note that the results for this initially textual language are in a much tighter range than for the graphical language in El Kouhen et al. [2]. The difference may be that this LWC task was heavy on generation and light on concrete syntax. The facilities for writing generators in the various tools may be rather similar in their productivity; at least their approach and structure is often similar, as has been noted for Eclipse's Xpand and MetaEdit+'s MERL [12]. In contrast, El Kouhen's experiment included no generation, and the BPMN language is heavy on concrete syntax. Our own experience suggests that concrete syntax takes roughly three times as long as abstract syntax when using MetaEdit+'s WYSIWYG vector graphics-based Symbol Editor; using the programming or XML-based concrete syntax definition found in many other graphical tools could significantly increase that ratio.

5. SUGGESTED EXPERIMENT DESIGN

There are many possible and useful experiments to be performed on language workbenches, but as stated before we will focus here on an experiment to compare language workbenches as different ways of producing a DSM tool for the same language.

5.1 Basis of comparison

As language workbenches have matured, feature coverage has become less useful as part of a general experiment: all tools should be able to cover the main features necessary for the bulk of a language. Feature coverage is still useful for a non-experimental comparison, and a particular feature may be investigated in its own experiment. In a fixed time experiment, feature coverage achieved could perhaps be used, but for any true comparison that would require the same order of feature implementation, and also equally sized features – a difficult task in general, rendered impossible by the differing processes and abilities of tools in different areas.

Lines of code seems fundamentally flawed as a method of comparison for the different and non-textual languages used for language definition. These problems are still prevalent but to a lesser extent in the languages for generator definition.

Time is in many cases directly the variable of interest for the implementation of a language and generators. It is also directly comprehensible by readers without knowledge of the workbench in question: if a task is known to take 25 days with GMF, the effort is clear; if it takes 1400 lines of code in Spoofax, the reader is unsure how much effort that requires compared to a more familiar language like Java.

Cost is an important factor in considering an overall DSM project, and even in choosing a single tool. It is not, however, an intrinsic property of a tool – prices for the same tool vary by customer and over time, and a previously commercial tool may even become free and open source, as has happened with at least two language workbenches. For an experiment in particular, cost would seem to have no meaningful role.

Time thus appears to be the most useful basis of comparison, which leaves the question of what time to measure and how to measure it to a good degree of trustworthiness: we shall return to these questions later in this section.

5.2 Users

Many of the previous comparisons have been performed by students. This seems an extension of the fallacy of “compare using what you can easily measure”: students are cheap, plentiful, and need teaching in this area anyway. However, most undergraduates are well below the level of the average language workbench user, so a full experiment is often beyond their abilities and a reasonable length, and even if performed will give results of dubious value. More targeted experiments on a particular feature may be possible, as in [5].

There is also the question of the practicality of the experiment: El Kouhen’s comparison required nearly 10 weeks of effort, and adding the last tool in would have added either 1% to the overall effort, or over 100%. If all that effort were to be expended by the same team, there would be a force pushing to leave slower tools out, or to reduce the experiment to something that may be too simple to give meaningful insight into real use. One option is to use resources from different sources, teams or projects – yet no team would want to be saddled with a randomly assigned yet unfairly sized share. The approach of LWC may be the best: the users would be people that are already associated with the tool, either as makers or expert users (e.g. consultants). They have a vested interest that encourages them to spend the time to have their tool included in the comparison, and to do the tasks well. Admittedly, poor performance in an earlier comparison may discourage a team from participating in future – but hopefully they would then use the saved effort to improve their tool.

Using experienced users gives more reliable and repeatable results, but does not measure the cost of learning. That would seem to be something that could be measured best separately (perhaps by an experiment with postgraduate students): it is an up-front cost to the first DSM project, but not subsequent projects. We can thus separate out the costs of various phases: learning the language workbench, creating a language with it, learning the resulting DSM tool, and creating a system with it. As mentioned, we focus here on language creation.

5.3 Tasks

Given the indications that effectiveness varies according to task both across tools and within a tool, it seems clear that we want to obtain figures for both the overall time for a tool, and for its performance on individual task areas such as abstract syntax, rules, concrete syntax, and generation or interpretation. This motivates breaking the tasks down into clearly separate phases, perhaps in a manner that would be unnatural in a real project: e.g. it is common to add at least basic concrete syntax for each new abstract syntax concept, even if a later phase would concentrate on improving all the concrete syntax.

For the most realistic setting, the tools would be given a domain description and the current code that is produced by hand (or whatever similar output is required). This was the case in the MDD-TIF07 workshop [13], the first comparison of language workbenches used by their makers or expert users. A down side of this approach was that the results were highly variable: some languages were poor in quality, and all were different. This still fulfilled the purpose of MDD-TIF – to familiarize tool makers with other tools – particularly where the tools showed how to create the languages from scratch in their presentation slot. it does not however lend itself to a quantitative comparison of language creation (although it could be useful if the extra step of measuring the productivity of the resulting language was included).

A more practical and targeted approach is to specify the desired modeling language, thus skipping the creative stage of inventing a language for a domain – probably more a test of DSM skill than the tool. Some latitude can be allowed for the tools to deviate from the language, although explanations should be given as to why: readers will thus be aware of the trade-offs a tool has made between a faithful rendition and an easier one.

For the output, experience with the Language Workbench Challenge [11] has shown that much of the work of implementation can end up being spent on creating a framework for the generated code to run on, if such is not provided. Similarly if the target platform is unfamiliar, significant effort is expended on learning it – again not related to the tool. It is thus best to specify the target platform, including which libraries etc. should be used, and also to provide an example model and its resulting reference implementation, so tools can just aim at generating that code.

5.4 Measurement

Measuring the full time for a real project is not possible in an experimental setting, nor is it believable if the developers or proponents of the tools are asked to record their own times. One option could be to reveal each individual task at a pre-set time, and have users submit their results online as soon as they are finished. The definition of “finished” is problematic, as work under such time pressure will most likely contain bugs, which would otherwise have been found and corrected at a later phase. More seriously, it is unlikely that the tool makers will be willing to commit ahead of time to these periods; a real customer’s problem may appear and will take precedence.

One solution could be not to record the time for the process of creation, but only for the creation of the finished result – in a similar way to how lines of code only measures the final code. Some information is certainly lost, but perhaps not critically, and it is at least in a familiar fashion. Users could complete the task in their own time, when convenient to them, and when finished they could record a video of creating the task from scratch. As well as providing an objective measure of that time, that would also serve as a tutorial for new tool users to learn the tool. This has the added benefit that any attempts to type or click as fast as possible are counter-productive: they might “win” the comparison, but they would certainly lose in getting new users to understand their tool. At least at this stage in the market, that is a paramount concern for all tools, commercial or otherwise.

Such an approach has already been successfully trialed in the first Language Workbench Competition, where each tool was given 40 minutes to present how to create the languages and generators. In practice only the MetaEdit+ presentation [14] did this from scratch, with other tools showing the resulting languages or parts of the language definitions that had been made earlier. A fixed presentation time was thus not appropriate, given the variation of speed of use of the tools. All tools however produced PDF tutorials showing how to create the languages. A video of the workbench completing the tasks would thus be possible and comparable for all, even if it could not all be sensibly presented in a workshop format.

Splitting the video into segments for each task will allow investigation of the relative and absolute strengths of each tool, as well as providing a more palatable tutorial for new users. Although the aim would be for all tools to complete all tasks, it would also improve the comparability if some tasks are omitted.

6. CONCLUSIONS

Domain-Specific Modeling has shown great promise for improving the productivity of software development, and language workbenches have shown great promise for efficient creation of DSM solutions. In spite of several workshops and comparisons, research shows that there is a wide range of effectiveness in the tools, tool makers often misjudge the actual value of features and approaches, and both tool makers and users are reluctant to look at tools from outside their own frame of reference – be that textual vs. graphical, Eclipse vs. Visual Studio, OMG standards vs. DSM-specific.

An objective, trustworthy comparison may go some way to help address these problems. Although achieving such a comparison faces significant challenges, not only technical but also those of personal or commercial interest, MDD-TIF and LWC have shown that the tool makers are willing to participate. El Kouhen et al.’s comparison also shows that new tool makers are ready to learn from what is already out there – something that has sometimes been lacking in the past – and that objective, quantitative results can be obtained.

Combining the best features of previous comparison designs allows us to avoid problems encountered earlier, and get closer to a comparison that will be of real value to users contemplating DSM, and also to current and future language workbench makers. So far, this is just a sketch; further work and a trial will be needed to flesh it out, and it must be combined with qualitative measures.

ACKNOWLEDGMENTS

Many thanks to the reviewers and the members of the Language Workbench Challenge mailing list, whose suggestions, criticisms and support contributed to this exploration. Opinions and failings of course remain solely those of this author.

REFERENCES

[1] Kelly, S., and Tolvanen, J.-P. 2008. *Domain-Specific Modeling: Enabling Full Code Generation.* Wiley.

[2] El Kouhen, A., Dumoulin, C., Gérard, S., and Boulet, P. 2012. *Evaluation of Modeling Tools Adaptation*. CNRS HAL. http://hal.archives-ouvertes.fr/docs/00/70/68/41/PDF/Evaluation_of_Modeling_Tools_Adaptation.pdf

[3] Kelly, S. 2004. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In *Proceedings of the OOPSLA & GPCE Workshop on Best Practices for Model Driven Software Development*. http://www.softmetaware.com/oopsla2004/mdsd-workshop.html

[4] Marttiin, P., Rossi, M. , Tahvanainen, V.-P., and Lyytinen, K. 1993. A comparative review of CASE shells: A preliminary framework and research outcomes. *Information and Management,* 25:11-31.

[5] Kelly, S., and Rossi, M. 1998. Evaluating Method Engineer Performance: An Error Classification and Preliminary Empirical Study, *Australian Journal of Information Systems* 6(1). http://dl.acs.org.au/index.php/ajis/article/download/316/283

[6] Isazadeh, H., 1997. *Architectural Analysis of MetaCASE: A Study of Capabilities and Advances*, Master’s Thesis, Queen’s University, Ontario, Canada. http://www.collectionscanada.gc.ca/obj/s4/f2/dsk2/ftp04/mq20654.pdf

[7] De Smedt, P. 2011. *Comparing three graphical DSL editors: AToM3, MetaEdit+ and Poseidon for DSLs*, University of Antwerp. http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/201011/projects/Philip.DeSmedt/report/report_PhilipDeSmedt.pdf

[8] Pelechano, V. Albert, M., Javier, M., and Carlos, C. 2006. Building Tools for Model Driven Development. Comparing Microsoft DSL Tools and Eclipse Modeling Plug-ins. In *Proceedings of the Desarrollo de Software Dirigido por Modelos - DSDM'06 (Junto a JISBD'06)*, Sitges (Barcelona) España. http://ceur-ws.org/Vol-227/paper11.pdf

[9] Özgür, T. 2007. *Comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks for Domain-Specific Modeling In the context of the Model-Driven Development*, Master’s Thesis, School of Engineering, Blekinge Institute of Technology, Sweden.

[10] Palarea, P. G., and Ramón, Ó. S. 2006. *Metamodeling Tools: Microsoft DSL Tools vs. MetaEdit+* [in Spanish]. http://dis.um.es/~jmolina/Pfc/DSLvsMetaedit.pdf

[11] Erdweg, S., van der Storm, T., Völter, M., Boersma, M., Bosman, R., Cook, W. R., Gerritsen, A., Hulshout, A., Kelly, S., Loh, A., Konat, G., Molina, P. J., Palatnik, M., Pohjonen, R., Schindler, E., Schindler, K., Solmi, R., Vergu, V., Visser, E., van der Vlist, K., Wachsmuth, G., and van der Woning, J. 2013. The State of the Art in Language Workbenches: Conclusions from the Language Workbench Challenge. In *Proceedings of the Software Language Engineering conference* (Indianapolis, USA, October 26-28, 2013). Springer.

[12] Kelly, S., and Kern, H. 2009. *Processing of MetaEdit Models with oAW*. http://www.metacase.com/blogs/stevek/blogView?showComments=true&entry=3428923761

[13] Völter, M., Gray, J., Kelly, S., and White, J. 2007. *Model-Driven Development Tool Implementers Forum*, Tools 2007. http://www.dsmforum.org/events/MDD-TIF07/

[14] Kelly, S. 2011. *MetaEdit+ LWC Demonstration*. Video/PDF. http://www.metacase.com/support/45/repository/#LWC