Collected Experience and Thoughts on Long Term Development of an Open Source MDE Tool

Lars Hamann, Frank Hilken, and Martin Gogolla

University of Bremen, Computer Science Department
D-28334 Bremen, Germany
{lhamann,fhilken,gogolla}@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. During 14 years of developing an open source model driven engineering tool at a university we collected some dos and don'ts for such projects, which we are going to describe in this paper. For example, the mentoring of students and afterwards the integration of their results need special treatments, to be able to keep a product of high quality. Beside such quality related issues, we also report on our experience with industrial cooperations. To get an idea about the amount of work that has been put into our tool, we review and visualize its history.

1 Introduction

In this paper, we report about the history of an open source software (OSS) tool used in the context of model driven engineering (MDE). Started as an implementation to validate formal semantics of the Object Constraint Language (OCL) that was defined in a doctoral thesis [14], ongoing development and maintenance continuously increased the tool features, while retaining a stable core. The acceptance of our tool USE (UML-based Specification Environment) [16] can be seen by the increasing times it was downloaded. Starting with ∼1,500 downloads in 2011, the number increased to ∼3,100 in 2012 and ∼4,700 in 2013. Further, several research projects apply or integrate USE into their own application, e. g., the TractsTool [3] integrates USE to be able to validate model transformations or the application of USE to describe and apply a domain specific language (DSL) for activity theory in requirements engineering [6]. To our knowledge, several universities also employ USE for education.

In this paper, we highlight some of the key extension that were made to USE during its 14 years of development. Due to the origin of USE as a research project, we further highlight some of the specific characteristics that apply to open source projects at universities, e. g., experience with work of students.

The rest of the paper is structured as follows: In Sect. 2 we show the different areas in which USE can be applied. In Section 3 we first describe the historical development of USE, followed by pointing out some key extensions. The section ends with different thoughts on key success factors. After taking a look to similar OSS tools published by working groups at universities in Sect. 4 we conclude.

2 Applications of USE

2.1 Modeling with USE

The main application of USE is to validate and verify UML models. These models are typically used during early stages of the software development process and focus on key parts of the designed system. This allows to exclude errors in the design at the very beginning. As it is commonly reported, e. g. in [2], requirements issues are cheaper to solve at the beginning of a software development process than at the end. If wrong requirements are implemented, a lot of rework has to be done. Further, these misunderstood requirements may lead to an inadequate architecture of the system, which further increases the overall costs. To be able to validate requirements, an early formal model, which covers main aspects of the system to be developed, can help to reduce the overall development costs. USE supports this validation by supporting the modeling with a well-defined subset of UML together with nearly complete support of OCL. Models in USE can be instantiated to be able to verify the specification at runtime without a complete implementation. This allows a modeler to incrementally extend the specification of a system on a formal basis without any need to make assumptions about the infrastructure that the system is going to be built with or runs on.

The most basic validation feature of USE is the static instantiation of a model. By this, we mean the creation of a system state without any dynamic behavior. In USE a system state consists of objects with their attribute values and links between them. Using such a state, a user can verify assumptions about the static specification, e. g., multiplicity, subsets, union, and redefines constraints. Further, it can be verified whether invariants are excluding the right states. The creation of system states can be done in various ways. For exploration of a model, the user can create objects and links in an ad hoc manner by using the graphical user interface (GUI). This is especially useful for bigger models, as the GUI can guide the user. For example, USE only shows commands for creating links between selected objects for which valid associations exists. To create and reproduce system states and to simulate dynamic behavior USE supports the execution of a command language. During simulation, specified pre- and post-conditions of operations as well as transitions of state machines are validated.

2.2 USE for Education

During our lectures about the design of information systems, we have successfully employed USE for more than 10 years now. One advantage of USE for education is the possibility to learn UML and OCL in an interactive way. Students can easily try different designs and get a rapid response about its correctness. For example, its fairly easy to create system states by dragging object instances and linking them to check defined constraints on the fly. Since USE can directly provide feedback about the current state of each constraint, one can immediately see, if the constraints are correct. This can be done by creating valid and invalid instantiations and by comparing the result of the evaluation of USE to the

developer's expectations. To a given extent, the possibility to completely define models in a declarative way together with the option to simulate scenarios fosters the ability to prescind.

2.3 USE as a Component

The integration of USE as a component into other products is supported by its two-level architecture. The user interface is decoupled from the system core, which allows other applications to easily integrate the UML and OCL validation capabilities of USE. This integration can be done in two different ways. Products can either implement interfaces required by the USE system or they can map their own implementations to USE instances. While the former approach reduces the memory consumption at runtime, because no additional instances need to be created, the implementation is more time consuming. Whereas, using the mapping approach reverts those pros and cons. Mapping application specific instances to USE concepts is supported by a recently added simplified application programming interface (API). In fact, it is an implementation of the facade pattern, which reduces the learning time for integrating USE as a library since internal details are hidden behind the facade. An example of an application, which integrates USE as a component is the so-called XGenerator. This model-to-text transformer is used in the context of defining data exchange standards for the German public authority [5].

3 More than 10 Years of Development: Lessons Learned

In this section we are going to take a brief tour through the history of USE. We highlight milestones and relate them to some metrics.

To get an overall impression on how the structure of USE has changed over the past 14 years of development Fig. 1 and 2 visualize the internal structure of USE as a city map [17]. Each class is represented as a building in these maps, using the number of operations as its height and the number of attributes as its width. Packages are shown as districts surrounding their members.

Figure 1 shows the city map of version 0.9.0 which was released in May 2000. This first version was build out of 489 classes spread over 27 packages with a total number of roughly 15,000 lines of code (LOC). The city map of the current development version of USE 3.1.0 is depicted in Fig. 2. As one can see, the city shape changed heavily over the last 14 years[1]. One example is the *downtown district* representing the GUI related classes that can be seen at the south of the city (see also Fig. 3). While the first versions of USE focused on the implementation of OCL (the eastern part of the city), little work was done on the graphical user interface. This changed over the versions, until the previously empty district was filled with GUI related classes. The currently not yet released version of USE (3.1.0) now contains some 86,000 LOC and nearly 1,600 classes distributed over 92 packages.

[1] An animation of the city maps of all released versions of USE can be found online: `http://www.db.informatik.uni-bremen.de/projects/use/CodeCity.gif`

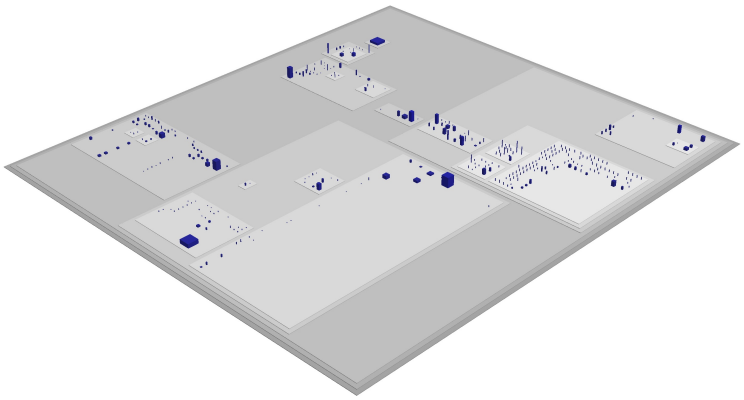

Fig. 1. CodeCity of USE 0.9.0

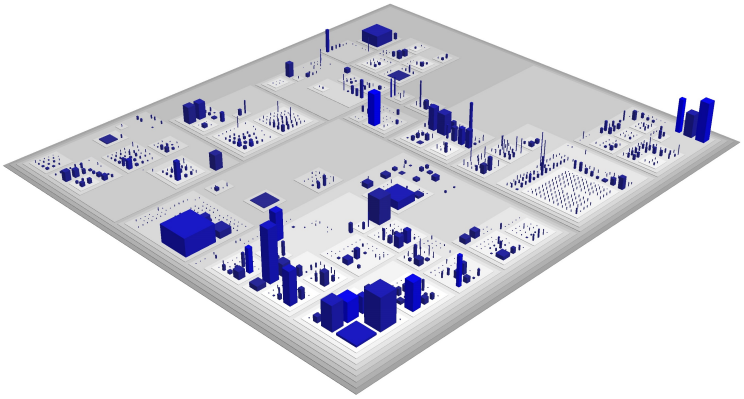

Fig. 2. CodeCity of USE 3.1.0

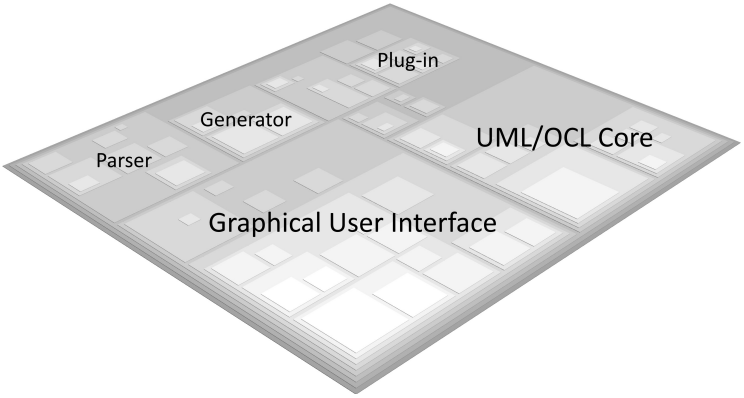

Fig. 3. Districts of USE

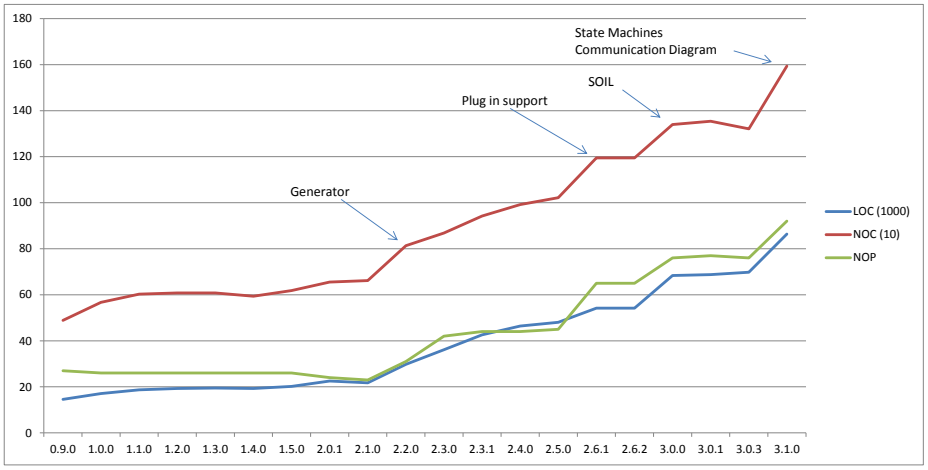

Fig. 4. Development of certain metrics of USE

3.1 Key Extensions

Figure 4 highlights the 14 years long history of USE by means of three metrics: (1) Lines of Codes (LOC), (2) Number of Classes (NOC), and (3) Number of Packages (NOP). Further, key extensions that were added to USE are highlighted. In the following, these extensions and selected plug-ins which are not covered by the given metrics are described briefly and – if appropriate – references to more detailed publications covering the described topic are given.

Ver. 0.9.0 - 2.0.1: OCL Core Through the years 2000 and 2001[2] the development of USE concentrated on the OCL core language up to version 1.4 of OCL. In parallel, a formal basis of OCL was developed and published as a doctoral thesis [14] in 2002. In the end, USE 2.0.1 supported static and dynamic validation as well as visualization of system states and scenarios by providing object and sequence diagrams.

Ver. 2.2.0: Generator A first verification feature was added in 2005. This generator allows to search for model instances using a language called ASSL (A Snapshot Sequence Language) [8]. ASSL is a procedural language with integrated backtracking support. One benefit of using such a language is the possibility to guide the search for valid model instances, which reduces the overall complexity. This extension was the result of a diploma thesis.

Ver. 2.6.1: Plug-in Architecture To support easier development of extensions for USE, a plug-in architecture was developed by a student during his thesis. Using plug-ins, USE can be extended and customized without touching the core. At the end of this section, three examples of such USE plug-ins

[2] Note, that the development of USE started some years before the first public version was made available.

are described. Note that the plugins do not count to either the city map or the statistics in Fig. 4.

Ver. 3.0.0: SOIL The foundation of another milestone of USE was defined formally by a doctoral thesis [4] in 2010 and implemented by a student for his diploma thesis. This extension introduced an action language called SOIL (Simple OCL-based Imperative Language). Using SOIL, a modeler can specify behavior using a well-defined imperative language that could for example be transformed to Java.

Ver. 3.1.0: State Machine and Communication Diagram Beside usability improvements, the current version of USE (2014), which is available as a preview, added support for defining and validating UML protocol state machines with full OCL support [10]. This allows for an easier way of defining behavior covering sequences of operation calls in a declarative way.

Plug-in: Model Validator Another approach of verifying models, has been added to USE by a plug-in called *model validator* [13], that was implemented by a student for his master thesis, too. In contrast to the previously described generator feature of USE, which applies a white-box approach for model finding, the model validator uses a black-box approach. Only few information, e. g., the number of instances for each class, needs to be provided for the finding process. Given this information, the plug-in translates given UML models including constraints into the relational logic of Kodkod [15]. KodKod itself uses SAT-solvers to find valid solutions that are transformed back into valid UML structures by the plug-in.

Plug-in: Monitor To apply USE for runtime verification, that is to verify a concrete implementation relative to its specification, the so-called USE-Monitor can be applied [9]. Supporting multiple target platforms using adapters, like the Java Runtime Environment and the Mircosoft Common Language Runtime, it can be used to connect to a running system and verify the current state against the model-based specification. Further, dynamic verification is supported by continuing the execution of the connected system, which allows, for example, the verification of protocol state machines.

Plug-in: Filmstrip To improve on the applicability of the model validator, a new plug-in has been added which performs model transformations from models with operations specifying their behavior into model validator compatible *filmstrip* models [11]. With these, it is possible to verify sequences of operation invocations using the otherwise static model validator.

3.2 Thoughts on Factors of Success

In this section we are going to describe what factors of success are, from our experience, required to build and maintain a successful open source MDE tool. We have split these factors into three groups, each covering different aspects of a long-term OSS development.

Technical One of the most crucial factors is, in our opinion, to provide a sound system. This means, it should be easy to set up, to execute, and to use. Not to

forget, it should be as stable as possible. To get and to keep a stable system, a leading architect who keeps an eye on the overall structure is essential. In the best case, this architect is something like a *geek*[3]. For our project, we can state that during the time there was someone in our working group who could be seen as a geek, the project made quantitative and qualitative progress, whereas if no such person was present, the development was nearly stuck.

On a technical level, tests are even more essential to get and keep a product of high quality. For USE, we run about 700 unit tests and over 120 different regression tests each time someone is committing changes into the source control system. Overall they compare nearly 1,200 expressions against expected results and execute more than 28,000 statements. Running all tests after each commit and notifying the development team about failed tests, like we do using a continuous integration server, further motivates the developers to fix errors as soon as they are encountered, which benefits the overall stability.

To improve the productivity of the development process and to reduce the likelihood of errors, it is a best practice to look for popular and active open source libraries as to built-up your system with. However, each new library introduces the burden of keeping an eye on new versions and even more important, it needs to be checked if the license of the component is compatible to the one under the own product is published. Because many open source licenses exist, each forcing different requirements and granting different rights, this is not a trivial task [7].

Organizational Although, USE is developed and maintained at a non commercial organization, economics play a role. To be able to increase the effort put into the project, it is quite a good idea to get an industrial partner who is in the need for the current or future product and supports the work financially. We made good experience even with long term cooperations. However, for research tools, one must be careful not to get into the role of a service provider. In other words, ongoing research and not only the maintenance of a product for the cooperation partner should be the goal.

Community and Feedback Even if software is open source, this does not mean you got a community of developers[4]. Therefore, by community we mean active users of our software. These can be modelers, who apply USE to gain information about their models, educators and students, who utilize it during lectures or researchers integrating USE as a component into their own tool.

For USE, we see two facts why it has reached such a high acceptance for research and industrial projects. First, publishing ongoing work at relevant conferences. Second, cooperations between our working group and commer-

[3] For an exiting view on geeks we kindly refer to question six from *Twenty Questions for Donald Knuth* (`http://www.informit.com/articles/article.aspx?p=2213858`, May 20, 2014)

[4] c. f. `http://readwrite.com/2014/07/07/open-source-software-pros-cons`

cial companies, like the integration of USE into the XGenerator as described earlier.
Regardless of the audience and as with commercial software, a profound support must be available. For our project, we run a discussion forum and a bugtracker at our project side on SourceForge[5]. Further, we provide information how to contact the developers in case of problems. For all of them, we try to keep response times low.
Nevertheless, publishing a tool as open source, even if no active developer community can be created, one can gain the benefit of receiving patches for issues found by users. While this is not the most common case, sometimes interested users already debug issues and propose fixes for them.
Another successful way how we collected feedback about new features and their value and usability was the usage of USE during lectures at university. The view of students as (usually) unexperienced users can help to identify possible usability issues, but can also guide the direction of development, i. e., whether it is worth to continue the work on a feature.

3.3 Thoughts on Extensions by Students

As we showed previously, some extensions to USE were made by students. Either for their diploma thesis, while participating in a collegiate project, which is mandatory at University of Bremen, or while working as a student assistant. As in the economy, the productivity and more importantly the quality of their work extremely differs. For many students, their thesis project is the first time they need to implement a bigger software component. These students need a well-delimited topic to work on and usually a more time consuming mentoring to achieve a good result. A kick-off meeting, introducing the topic and to give a first insight of the relevant part of the system should be obligatory. When accompanying them during their time of writing their thesis, one often gets usable results. However, the lack of experience often leads to working but poorly maintainable code. The usage of a best practice guide can help to improve the code quality, but commonly one has to refactor certain parts of the produced results. For USE, we started to create such a best practice guide, since we noticed recurring bad smells.

One example of such bad smell is the tendency of students to extract information out of strings (if you are lucky they won't use the identity comparator (`==`) to check equality of strings). This leads to issues hard to identify if later on string representations change. While the wrong usage of the identity comparator on strings in Java can automatically be discovered by tools such as FindBugs[6], the extraction of information out of strings cannot. For this, we tell students: "if you start to write something like `aString.indexOf(something)`: rethink!".

Another point, that is worth noticing, is that unexperienced students have fun reinventing the wheel. It seems to be a pleasure implementing all the cool

[5] `http://www.sourceforge.net/projects/useocl`
[6] `http://findbugs.sourceforge.net`

algorithms for sorting and searching learned during their study. Unfortunately, most of the times these algorithms either contain bugs or do not perform well. For this, at the beginning of their work a hint at already used libraries reduces the likelihood of the tenth sorting algorithm in the source code.

Last but not least, students should be encouraged – if not forced – to write tests containing more than tiny amounts of data. Otherwise the main developers of the extended system are going to spend a lot of time resolving bottlenecks.

4 Related Projects

Several long running open source MDE tools exist. Unfortunately, we can highlight only two of them, because of space restrictions. Dresden OCL [1] is, in contrast to USE, an Eclipse/EMF-based OCL interpreter which is also integrated into several modeling tools. Its development started 1999[7] at the Technical University of Dresden, where it is still maintained. As USE, Dresden OCL is extended mainly by university members.

The Alloy Analyzer[8] uses the Alloy language [12], which is based on the notion of relations. Alloy models can be used to find valid instances, w. r. t., defined constraints, but can also be used to find counterexamples. The Alloy Analyzer, like the USE ModelValidator, applies Kodkod [15] to search for valid model instances. The Analyzer and the language are developed at the MIT.

5 Conclusion

In this work we have shown, how an open source MDE tool that was the outcome of a single doctoral thesis emerged over more than 14 years of development. We described key extensions that were made by university members and discussed factors for the development that are in our view important for a successful long-term evolution of an open source tool. An anecdotal list of issues we encountered while integrating extensions made by students and the proposed best-practice guide, can help projects to receive better results when unexperienced developers are possibly temporarily working on a project.

Acknowledgments We would like to thank all the people who have contributed to USE over the years. In particular: Fabian Büttner, Mirco Kuhlmann, Mark Richters; Roman Asendorf, Jörn Bohling, Jens Brüning, Torsten Humann, and Hendrik Reitmann

References

1. Aßmann, U., et. al.: Dropsbox: the dresden open software toolbox - domain-specific modelling tools beyond metamodels and transformations. Software and System Modeling 13(1), 133–169 (2014)

[7] http://www.dresden-ocl.org
[8] http://alloy.mit.edu/alloy/index.html

2. Boehm, B.W., Papaccio, P.N.: Understanding and controlling software costs. IEEE Trans. Software Eng. 14(10), 1462–1477 (1988)
3. Burgueño, L., Wimmer, M., Troya, J., Vallecillo, A.: Tractstool: Testing model transformations based on contracts. In: Liu, Y., Zschaler, S., Baudry, B., Ghosh, S., Ruscio, D.D., Jackson, E.K., Wimmer, M. (eds.) Demos/Posters/StudentResearch@MoDELS. CEUR Workshop Proceedings, vol. 1115, pp. 76–80. CEUR-WS.org (2013)
4. Büttner, F.: Reusing OCL in the Definition of Imperative Languages. Ph.D. thesis, Universität Bremen, Fachbereich Mathematik und Informatik (2010)
5. Büttner, F., Bartels, U., Hamann, L., Hofrichter, O., Kuhlmann, M., Gogolla, M., Rabe, L., Steimke, F., Rabenstein, Y., Stosiek, A.: Model-Driven Standardization of Public Authority Data Interchange. Science of Computer Programming (2013)
6. Georg, G., Troup, L.: Experiences developing a requirements language based on the psychological framework activity theory. In: Cabot, J., Gogolla, M., Ráth, I., Willink, E.D. (eds.) OCL@MoDELS. CEUR Workshop Proceedings, vol. 1092, pp. 63–72. CEUR-WS.org (2013)
7. German, D.M., Hassan, A.E.: License integration patterns: Addressing license mismatches in component-based development. In: Proceedings of the 31st International Conference on Software Engineering. pp. 188–198. ICSE '09, IEEE Computer Society, Washington, DC, USA (2009)
8. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4), 386–398 (2005)
9. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Cabot, J., Clariso, R., Gogolla, M., Wolff, B. (eds.) Proc. Workshop OCL and Textual Modelling (OCL'2011). ECEASST, Electronic Communications (2011), http://journal.ub.tu-berlin.de/eceasst/issue/view/56
10. Hamann, L., Hofrichter, O., Gogolla, M.: Towards Integrated Structure and Behavior Modeling with OCL. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012). pp. 235–251. Springer, Berlin, LNCS 7590 (2012)
11. Hilken, F., Hamann, L., Gogolla, M.: Transformation of uml and ocl models into filmstrip models. In: Ruscio, D.D., Varró, D. (eds.) ICMT. LNCS, vol. 8568, pp. 170–185. Springer (2014)
12. Jackson, D.: Alloy: A logical modelling language. In: Bert, D., Bowen, J.P., King, S., Waldén, M.A. (eds.) ZB. LNCS, vol. 2651, p. 1. Springer (2003)
13. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) Proc. 49th Int. Conf. Objects, Models, Components, and Patterns (TOOLS'2011). pp. 289–305. Springer, Berlin, LNCS 6705 (2011)
14. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. Ph.D. thesis, Universität Bremen, Fachbereich Mathematik und Informatik, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)
15. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS. LNCS, vol. 4424, pp. 632–647. Springer (2007)
16. A UML-based Specification Environment. Internet, http://sourceforge.net/projects/useocl/
17. Wettel, R., Lanza, M.: Visual Exploration of Large-Scale System Evolution. In: Hassan, A.E., Zaidman, A., Penta, M.D. (eds.) WCRE. pp. 219–228. IEEE (2008)