Key Properties for Comparing Modeling Languages and Tools: Usability, Completeness and Scalability

Timothy C. Lethbridge

Department of Electrical Engineering and Computer Science, University of Ottawa
800 King Edward Ave, Ottawa, ON Canada
tcl@eecs.uottawa.ca

Abstract. We discuss key properties that we have focused on when designing the Model-Oriented Programming language Umple and its support tools. Usability is key because we want to enable modellers to work rapidly, make fewer mistakes, and appreciate the benefits of modeling better. Completeness in code generation with respect to UML is important because most tools omit important semantics in the code they generate. Scalability is important because we want it to be possible to develop a system of arbitrary size. In this paper we highlight some of the essentials of how we have addressed these key properties in Umple. We also discuss how these properties can be used to compare modeling approaches.

1 Introduction

In this paper we discuss three important properties we have addressed in the development of the Umple model-oriented programming language and related technology. These properties are, 1) Usability, because many modeling tools are complex and slow; 2) Scalability, because graphical modeling languages in particular have issues in very large systems; and 3) Completeness, because most modeling tools generate incomplete code and many do not meet the completeness needs of users.

We propose assessment criteria for these and suggest these features be central to the comparison of modeling tools in general because they have such an impact on the usability and usefulness of such tools. They are among the issues being discussed at the Comparing Modeling Approaches series of workshops [13] [14].

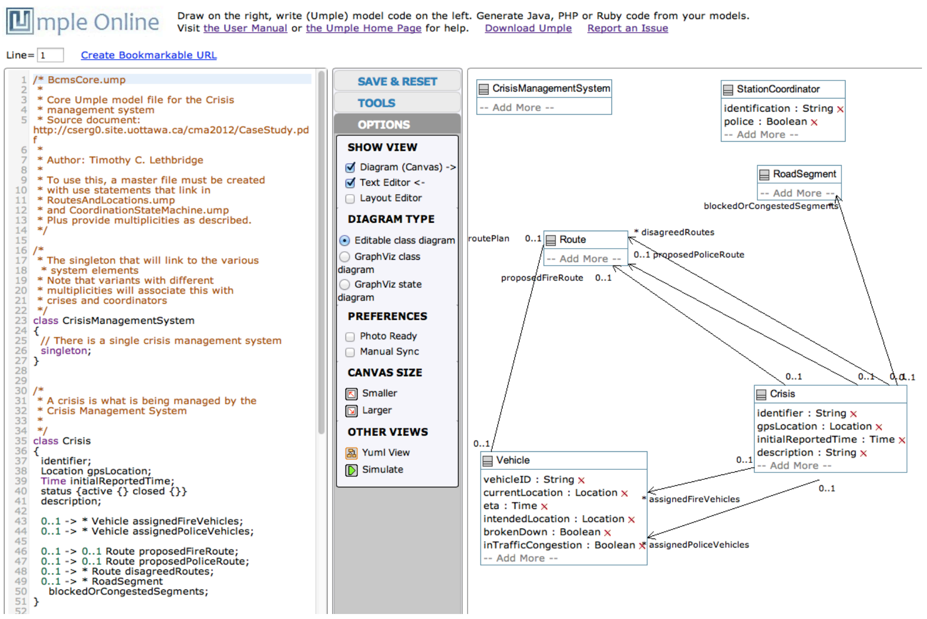
Umple [1] is a set of extensions that can be applied to C-family languages such as Java, C++ and PhP (we refer to these as base languages). Some Umple features are:

- It represents UML *associations* (specifications of sets of links between classes) as a language primitive, allowing a large amount of code for these to be generated. The generated code supports referential integrity, as in databases.
- It represents UML *state machines*, including arbitrarily nested states, concurrent regions, actions, activities run in separate threads, and various other features.
- It includes certain *patterns* such as immutable and singleton as primitives.
- It includes various *separation-of-concerns* abilities such as mixins and aspects.

- *Methods* in base languages are embedded in Umple code and passed to the Umple compiler unchanged; the remainder of the system's code is generated by Umple.

Umple merges programming and modeling, since Umple's textual form is the primary way in which a system is expressed. In this paper and in Umple, when we refer to modeling, we are talking about the use of abstractions that are not present in standard programming languages, such as state machines, associations, use cases, etc. The use of such constructs should allow the modeller to work at a 'higher level' when developing systems, as compared to the typical programmer. Umple, however, envisions that all such abstractions should conceptually be capable of being expressed in the same manner as in programming languages, and that many of them will in time become part of programming languages.

Umple provides a command-line compiler, an Eclipse plugin and UmpleOnline [2] (see Figure 1), a web-based tool. The latter is useful for education and demonstrations. All three tools support code generation; UmpleOnline also supports interactive editing in which a UML diagram is kept synchronized with the Umple text. Figure 1 shows a view of the bCMS system in the process of being edited by UmpleOnline. The example itself can be accessed online [3].



Draw on the right, write (Umple) model code on the left. Generate Java, PHP or Ruby code from your models.
Visit the User Manual or the Umple Home Page for help. Download Umple Report an Issue

Line= 1 Create Bookmarkable URL

```
/* BcmsCore.ump
 *
 * Core Umple model file for the Crisis
 * management system
 * Source document:
http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pd
f
 *
 * Author: Timothy C. Lethbridge
 *
 * To use this, a master file must be created
 * with use statements that link in
 * RoutesAndLocations.ump
 * and CoordinationStateMachine.ump
 * Plus provide multiplicities as described.
 */

/*
 * The singleton that will link to the various
  * system elements
 * Note that variants with different
 * multiplicities will associate this with
 * crises and coordinators
 */
class CrisisManagementSystem
{
  // There is a single crisis management system
  singleton;
}


/*
 * A crisis is what is being managed by the
 * Crisis Management System
 *
 */
class Crisis
{
  identifier;
  Location gpsLocation;
  Time initialReportedTime;
  status {active {} closed {}}
  description;

  0..1 -> * Vehicle assignedFireVehicles;
  0..1 -> * Vehicle assignedPoliceVehicles;

  0..1 -> 0..1 Route proposedFireRoute;
  0..1 -> 0..1 Route proposedPoliceRoute;
  0..1 -> * Route disagreedRoutes;
  0..1 -> * RoadSegment
    blockedOrCongestedSegments;
}
```

SHOW VIEW
- [✓] Diagram (Canvas) ->
- [✓] Text Editor <-
- [ ] Layout Editor

DIAGRAM TYPE
- [● Editable class diagram]
- ○ GraphViz class diagram
- ○ GraphViz state diagram

PREFERENCES
- [ ] Photo Ready
- [ ] Manual Sync

Fig. 1. Image of UmpleOnline showing part of the bCMS Umple model [3]

Key motivations for developing Umple were:

- Our research showed that existing UML Modeling tools were weak in terms of both usability [4] and completeness of code generation [5] [6]. We wanted to develop a tool that would be accessible to the open source community [7], which has always preferred to develop in a code-centric manner. Neither ArgoUML [8] nor Papyrus [8], both prominent open source modeling tools, are developed by having their code generated from models. They are coded in a traditional manner. We wanted to break this trend by developing Umple in itself.

- We wanted a low-footprint modeling tool that we could use in the classroom.

In the following sections, we discuss usability, completeness and scalability.

2 Usability

Given the key motivations behind Umple, as discussed above, it was central that usability should be our central concern in the development of Umple or any other modeling tool. We have focused on the following aspects of usability that we believe are central, and propose these as sub-criteria for comparing modeling approaches.

It should be easy to install: A problem with many tools is that they are not 'instant-on'. This hampers not only use in the classroom, but also spontaneous discussions among developers (such as 'whiteboard-style' sessions) or quick resumption of work after an interruption. In Umple, users can do quick work on the web using UmpleOnline, and have access to a command-line tool that also requires hardly any installation.

We propose this should be assessed on the following scale: i. Instant-on (typically web based); ii. Download-and-run; iii. Download and run with other components that must be pre-installed (such as an IDE), and which must be at this level or better in installability. iv. Complex to install (e.g. requires compiling, configuring, etc.)

It should be simple to learn: The tool must have modes that are extremely simple, so a beginner can instantly intuit what needs to be done. In Umple we have focused on making the language straightforward: Class declarations look like declarations in other languages; association declarations look like UML; state machines are just nested named C-style blocks. Similarly, at startup, UmpleOnline displays a central menu highlighting key actions: Load/Draw/Generate. There is no complex set up of 'projects' as is normal in other modeling tools or IDEs.

In Umple, we have chosen to allow models to be written with the least amount of special symbols, bracketing and other complex syntax. We have also ensured its syntax aligns with other C-family languages to ensure familiarity and ease of embedding. For graphical languages, Moodie provides a comprehensive set of ideas [15].

Assessing learnability can only be fully done with empirical studies, but to achieve top learnability there would need to be: a) Some examples that can be tried out with essentially no effort; b) Comprehensive feedback to assist users when they make errors or to suggest alternatives; c) Ability to do basic modeling by looking at and emulating or editing examples; d) A comprehensive user manual; e) Defaults for configurable items, so the user can 'get going' with a basic project right away. Undoubtedly this list of criteria will need to be extended.

It should be efficient and quick to use: Users should be able to edit models as fast as they can with the fewest mouse movements or keystrokes. We have found that for some tasks (such as adding a diagram element), two-click interactions that involve clicking on an icon, then clicking on the place in a diagram to add the element, work best. Renaming by clicking on a name and editing, can work well on a diagram too. However, for anything more complex, well-designed textual notations become superior. In Umple we chose to deploy tools that allow textual editing of UML models in a C-family syntax, alongside diagram editing. This gives the best of both worlds to users, and our surveys show that users are satisfied [10].

To assess this criterion, empirical studies would once again be needed. Measurement could entail determining users' ability to create benchmark models, to make specific changes to such models, and to answer specific questions about such models.

It should guide users away from errors: We observed that many modelers in industry produce models that are incorrect or incomplete [11]. Having comprehensive code-generation is one solution to this, since without code generation where the results can be tested, it is hard to receive feedback about models. Umple works like any other compiler, and provides many different warnings and errors that guide the user to create correct models. We have created a manual page that shows what causes the error, and gives an example of how to solve it. All errors, even in embedded base-language code, give pointers to the original Umple source, eliminating 'round-trip-engineering' which is error-prone and complex.

To compare tools there would need to be empirical assessment, measuring the frequency of errors, and speed of recovery from errors, when working on standard problems.

3 Scalability

Many modeling tools have scalability problems. We propose that scalability should mean the following for a modeling language or tool.

It should be able to model systems of arbitrary size and manage those models without slowing down: In Umple we have compiled systems of many thousands of lines, including Umple itself, or JHotDraw [12]. It should have acceptable response time when the system gets larger: Since Umple is text-based, editing large models is not an issue: integrated development environments have for decades been able to handle large multi-file systems. In addition, the Umple compiler can compile a very large system in not much more time than a comparable Java system can be compiled.

To assess this, measurement of editing speed, and, where appropriate, analysis speed or code generation speed would be needed. Any slowdowns for editing of massive models would show lack of scalability, as would non-linear slowdown of analysis or code generation.

It should have mechanisms for separation and concerns: These allow the user to work with portions of the system, and not get confused by overwhelming complexity. An Umple system can be divided into files in many ways: By class, or by feature for example. Pieces of a class can be defined in different files.

Having such mechanisms does not immediately guarantee scalability, since the modeler must still organize their model. However, lack of such capability does limit scalability.

To assess this criterion, modeling approaches should be tagged according to whether submodels can be edited and analysed independently of the larger model, and whether any of the following techniques are employed: Composition of multiple elements, aspect orientation, mixins/traits, variants, and tools to show the relationship among the model components (e.g. generated UML package diagrams) as well as edit such relationships.

It should have search tools allowing finding of elements in a complex model. Since Umple is a textual modeling tool at heart, the full power of existing textual search tools can be used.

Graphical modeling approaches should be assessed based on how easy it is to search for elements or structures.

4 Completeness

Completeness in a modeling tool can be assessed along a number of dimensions:

- Completeness with respect to a standard language (e.g. UML).
- Completeness with respect to user needs, as assessed by empirical studies.
- Completeness of analytic abilities, i.e. the extent to which possible analyses of the semantics of the model are performed to find errors or inconsistencies.
- Completeness of generation, i.e. whether all relevant semantics expressed in the model is reflected in generated code. This only applies to languages that are intended to generate code, and not to languages for such aspects as requirements.

In Umple we specifically avoided implementing all of UML, since a key criticism of UML is that it is so complex. Instead we have focused on the second criterion above: We have asked users to develop systems and when they have found significant shortcomings in Umple we have improved Umple by adding needed features. We continue to also address the third criterion by adding more layers of analytic capability. Umple already has a comprehensive set of warning and error messages, pointing out inconsistencies. We also have tools to generate metrics, and are working on interfacing Umple to formal tools for model checking and theorem proving.

Our previous research found serious gaps in the code generated by almost all code generators we were able to access. For example, most code generators for UML associations, including ArgoUML, simply generate a couple of variables in each class. The user has to write code to manipulate those variables. Umple generates code that accurately handles all constraints, as well as referential integrity.

Similarly, very few state machine generators do a proper job with nested states and concurrent regions. This is akin to having a word processor that doesn't print many types of documents. In Umple we have focused on ensuring that our code generation covers the complete semantics of the underlying formalisms to a much greater extent than other tools. This is documented in the theses of Badreddin [6] and Forward [5].

5 Conclusions

We have highlighted a number of key qualities of modelling tools that we feel should be the focus of attention for language and tool developers, and which should be used when comparing modeling approaches. We have pointed out a few ways we have addressed these in Umple, and have suggested how they could be assessed.

6 References

1. Umple home page, http://www.umple.org
2. Umple Online, http://try.umple.org
3. BCMS Core in Umple Online, http://try.umple.org/?example=BcmsCore
4. Forward, A., and Lethbridge, T.C. (2008), "Problems and Opportunities for Model-Centric Versus Code-Centric Software Development: A Survey of Software Professionals", Workshop on Modeling in Software Engineering, ICSE 2008, Leipzig, ACM, pp. 27-32
5. Forward, A., 2010. The convergence of modeling and programming: Facilitating the representation of attributes and associations in the Umple Model-Oriented programming language. Ph.D. thesis, University of Ottawa.
6. Badreddin, O, 2012, A Manifestation of Model-Code Duality: Facilitating the Representation of State Machines in the Umple Model-Oriented Programming Language, Ph.D. thesis, University of Ottawa.
7. Badreddin, O., Lethbridge, T.C., Elassar, M., 2013, Modeling Practices in Open Source Software, Open Source Software: Quality Verification, Vol. 404, pp. 127-139,
8. ArgoUML, http://argouml.tigris.org
9. Papyrus, http://www.eclipse.org/papyrus/
10. Lethbridge, T. C., Mussbacher, G., Forward, A., Badreddin, O., 2011. Teaching UML using Umple: Applying model-oriented programming in the classroom. CSEE&T, 421-428.
11. Farah, H. and Lethbridge, T.C., 2007, Temporal Exploration of Software Models: A Tool Feature to Enhance Software Understanding, WCRE 2007, Vancouver, IEEE-CS
12. JHotDraw, http://www.jhotdraw.org
13. Mussbacher, G. et al., 2012, "Assessing composition in modeling approaches", CMS '12 http://dl.acm.org/citation.cfm?doid=2459031.2459032
14. Georg, G. et al. 2013, "Modeling Approach Comparison Criteria for the CMA Workshop at MODELS 2013", http://cserg0.site.uottawa.ca/cma2013models/ComparisonCriteria.pdf
15. Moodie, D.L. 2009, "The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering", IEEE TSE, 35, 6, pp. 756-779