

A Novel Approach to Versioning and Merging Model and Code Uniformly

Omar Badreddin, Timothy C. Lethbridge, Andrew Forward
University of Ottawa, 800 King Edward, Ottawa, Ontario, Canada
obadr024@uottawa.ca, tcl@eecs.uottawa.ca, aforward@gmail.com

Keywords: Integrated environments, UML, Modeling, coding, version control, model merging and versioning, Model Drive Development.

Abstract: Model Driven Architecture (MDA) advocates the use of models, rather than code, as the main development artifact. Yet model versioning and merging tools still lag in capabilities, ease of use and adoption relative to source code versioning and merging tools. This forces many teams to avoid model-based collaboration and concurrent model modifications. In this paper, we highlight the main challenges behind the relatively small adoption of model merging approaches. We present a novel model-based programming technology that addresses many of those challenges. The approach treats code and models uniformly, effectively enabling modelers to version and merge models using existing text-based technologies.

1 INTRODUCTION

Software models, in addition to their usefulness in design and code generation, play an indispensable role in collaboration and communication. The trend towards increased software complexity, particularly larger and more complex models, emphasizes the need for model-based collaboration. Important aspects of collaboration include conflict resolution, as well as artifact comparison, merging and versioning.

Models are typically rendered visually using a language like UML, but stored and transferred using XML-based formats like XMI. This approach gives rise to complications for merging and versioning, since simple changes in the visual model tend not to map straightforwardly to simple changes in the storage format. Considerable research in the field of model-based collaboration therefore focuses on developing tools to adequately version and merge models. Unfortunately these have seen relatively little uptake, and have other important limitations.

This situation can be contrasted to source code versioning and merging. The latter domain has mature and feature-filled tools that are both widely adopted, and are very positively perceived, even in large teams with frequent changes and regular merging of conflicting changes. Research has shown that software developers have high confidence in

automated merging of code segments (Adams et al, 1986).

Early source code collaboration products like RCS and Visual Source Safe relied on pessimistic locking, requiring users to first secure all required software artifacts prior to making the necessary changes. The pessimistic approach can be usable for small teams and small projects, however it can be cumbersome for the developer to be in the middle of a change and discover the need to change one additional file that turns out to be locked by another developer. Pessimistic approaches to locking severely limit collaboration and flexible evolutionary development of software.

More optimistic locking approaches emerged in which files do not need to be exclusively locked prior to editing. These were developed based on the observation that change sets required to implement distinct fixes or features of a system do not often overlap; even within the same source file. And, when conflicts do arise, intelligent automatic merging works most of the time. Tools that support optimistic locking include SVN, Git, and CVS.

In an optimistic environment, where all resources are available for editing, the onus of maintaining consistency between versions is placed partially on the tooling's merge capabilities. In scenarios when conflicts cannot be automatically resolved, manual intervention is required.

See http://www.modelsward.org/Abstracts/2014/MODELSWARD_2014_Abstracts.htm

In this paper we present a novel approach to model collaboration: we align visualizing models with their textual representation using the Umple platform. Umple is a model-oriented programming language that supports modeling using a textual notation just like other high-level programming languages. Umple blurs the distinction between code and modeling, enabling modellers and coders to collaborate on models in a way similar to code-based collaboration. This is achieved without loss of the added value of the visual representation of models, since Umple tools can display a diagram unambiguously from the Umple code, and allow edits to such diagrams to automatically be reflected in the code. The Umple paradigm leverages existing textual merging techniques, and facilitates the transition of code-centric teams towards the use of more model-centric approaches.

This paper is organized as follows. We first analyze the barriers to adoption of emerging model versioning and merging techniques. In section 3, we introduce the Umple modeling and coding paradigm and show how it can support uniform versioning and merging of code and model artifacts. We then compare and evaluate our approach in terms of usability of model based versioning and merging.

2 KEY PROBLEMS: BARRIERS TO ADOPTION OF MODELING AND MERGING

Our prior research indicates that modeling practices are not as widely adopted as might be desired (Forward et al, 2010). The open source community, for example, universally uses code-centric approaches to collaboration and simultaneous code edits (Badreddin et al, 2013). Practitioners have high confidence and familiarity with existing code-merging and versioning technologies.

Existing approaches and tools for versioning and merging models have failed to catch up with the adoption of code-based versioning and merging. The following subsections describe challenges with existing approaches that might cause this slow adoption.

2.1 Heavy Reliance on Subjective Conflict Resolution

When a conflict occurs in model (or source code) merging, it is desirable to minimize user

intervention. Problems resulting from user intervention include:

1. The user's decision may be inconsistent with the intention of the original modification;
2. The user's subjective judgment is inevitably not uniform throughout the project life cycle which can bring unpredictability in model evolution;
3. Conflicts may arise that tools cannot readily handle, forcing the user to commit a change without careful analysis, simply so his or her work can be saved in the repository.

The third point is particularly important to consider: Pottinger and Bernstein (Pottinger & Bernstein, 2003) categorize conflicts based on the meta-level at which they occur. They identified three categories: representation, meta-model, and fundamental conflicts. An example of a meta-model violation would be merging two class diagrams, resulting in a subclass being also a superclass of itself, i.e. violating the strict ordering of the inheritance hierarchy. The capability to temporarily support such violations can be very useful, since it prevents 'forced' decisions and allows models to evolve in the series of steps most convenient to the modellers, even though intermediate steps may be unsupported by the underlying meta-model. In the above example, two different modellers may have independently created generalizations between two classes, but each chose a different class to be the superclass. A tool should be able to merge the changes and store the result regardless of the conflict. The tool should then be able to point out the inconsistency to the developers, who can debate which should in fact be the superclass, and then once the decision is settled, commit a change that resolves the conflict.

Allowing meta-model violations is not easily supported by graphical modeling tools. In the case of RSA (IBM, 2009), the compare facility has no support at all for temporary violations. Merging of changes that result in a violation is prohibited, forcing a modeller to artificially resolve a conflict before committing a new version.

2.2 Handling of Layout Information

A key feature of effective modeling is separation of the model itself from the (possibly many) diagrams that offer views of the model. This notion of multiple separate views is largely absent in source-code, but this added complexity of multiple views is a common occurrence in software models and is of particular importance to modeling. For effective

versioning and merging, separate consideration of model entities from diagram layout directives is necessary to reduce the number of potential merging conflicts. Adams et al (Adams et al, 1986) proposed a version control system where conflicts in diagram element coordinates can be ignored. The only analogy in source code is the need to ignore cosmetic changes such as indentation and whitespace in order to reduce the number of merging conflicts.

Historically, XMI-based tools for model merging did not allow separation of layout information, adding considerable complexity to merging. This situation is changing with the gradual adoption of Diagram Interchange standards in tools such as Papyrus (OMG, 2012). Nonetheless, model-merging tools continue to face a challenge because of the presence of multiple diagrams, and the fact that the diagrams are exchanged using a separate notation.

2.3 Adoption of Model Merging Techniques

A number of model merging and versioning tools have been successfully implemented. However, their applications have been limited to specific industrial settings, such as (Adams et al, 1986). Wide adoption and standardization of these tools has not yet been achieved. Most available commercial tools rely on XML-based merging. For example, the latest releases of Borland Together, MagicDraw, and RSA continue to rely on XMI serialization for model versioning and merging. This aspect is further discussed in Section 2.5.

For a model-versioning technique to be widely adopted, the additional computational and infrastructure complexities of the technique should be kept at a minimum. In addition, the technique has to be easily extendable to other modeling notations.

2.4 Synchronizing among Related MDA Artifacts

Merging different versions of models brings about additional challenges for MDA. It is often the case that models have closely-related artifacts, such as other models, generated code, or hand-written code. Related artifacts of a model should be properly handled when merging model versions. Take, for example, a class diagram and a state machine diagram for a particular class in that diagram. In some tools, these two models are stored and versioned as separate and unrelated artifacts. It would be beneficial to allow versioning of both the

state and the class diagrams to be managed collectively or separately, if desired.

Here, three aspects of the versioning approach are crucial: a) the ability to easily and consistently extend the same approach to a number of modeling notations b) the ability to manage a number of models as a single artifact, or as separate but closely related artifacts c) finally, the ability to manage implications to generated artifacts.

2.5 Efficiency of Versioning and Merging

Existing approaches are not efficient for large models (Treude et al, 2007). Treude reports that medium to large size models can take five minutes to one hour of processing for merging. Some tools implement approaches that result in significant memory and computational overheads, for example, the use of Universal Unique ID (UUID) for all modeling elements, which cannot change once created, and whose uniqueness must be guaranteed at all times (Adams et al, 1986). It is common for those UUIDs to be more than 40 characters in length, and they are usually combined with the namespace of the modeling element. Such approaches are inevitably computationally complex, and limit extensibility to models created in different tools. This is comparable to preventing coders from merging code written in Eclipse with code written on a text pad.

On the other hand, XML processing is relatively efficient compared to semantic model merging. Most tools, such as ArgoUML (ArgoUML, 2009), make available an XMI interface, typically used with a version control tool. In particular, the following modeling tools have interfaces to repositories like CVS to enable XMI-based versioning: EclipseUML, Astah (Astah, 2009) and Poseidon (Gentleware, 2013). Although the computational processing of XML-based documents may be efficient, XML-based versioning gives rise to a variety of problems. In particular, XML does not have the appropriate level of abstraction, resulting in many false deviations (Altmanninger et al, 2008). XML generated from diagrams and models also tends to redistribute the modeling elements in unpredictable ways, meaning that a very small change that a software engineer makes can either result in a considerable number of small changes throughout the XML file, or worse, an XML file that is arranged quite differently. This is a central problem for merging and versioning tools that relies on XMI for versioning control.

2.6 Non-determinism and unpredictability

Model merging tools need to be deterministic so that conflict resolution can be predictable and reversible. SIDIFF (Treude et al, 2007), for example, creates a directed typed graph from the XMI model where similar nodes are identified by giving weights to their attributes. This approach is non-deterministic because it depends on the weights given for the specification of level of similarity between modeling elements. In addition, similarity itself is a factor that changes over the lifetime of the project. For example, location proximity on a diagram is usually taken into consideration when determining how ‘similar’ two modeling elements are. Location proximity is a factor that changes over time, rendering the merging tool non-deterministic.

Even when determinism is achieved by storing users’ merging choices, the model’s evolution can easily become unpredictable, since users choices can vary significantly.

3 THE UMPLE SOLUTION

We developed Umple initially as a way to bridge the model-code divide. In other words, it was designed with the primary intent to add all important modeling concepts to programming languages so programmers can model, and also so that modellers can use programming language formalisms. An important additional benefit of Umple however, is that it brings the elegant power of source-code versioning to modeling ‘for free’.

Our tool suite has the following main features:

- It provides a high-level textual abstraction of modeling elements and relationships.
- It provides a textual abstraction of layout information.
- It separates the layout information from the modeling elements.
- It weaves together modeling abstractions with algorithmic code for uniform versioning and merging.

Umple is particularly suitable for software teams that currently collaborate using source code, and wish to adopt modeling. In some cases, teams have had difficulty adopting modeling due to the complexities of merging and versioning. Umple does not require new versioning infrastructure, but rather, makes use of existing infrastructure available for

code versioning. In the remainder of this section, we present our tool and report on our experiences.

3.1 Umple Tools for Model Versioning and Merging

Umple is model-oriented programming language with a suite of tools that supports all class diagram modeling elements (classes, attributes, associations, multiplicities, role names, comments, inheritance, association classes) and most of state machine modeling notation (states, transitions, entry and exit actions, transition actions, do activities, nested states).

Umple integrates modeling artifacts mentioned above with action languages; currently supporting a Java-based, Php-based and C++-based implementations. Because the textual representation of Umple uses syntax that is similar to these languages, model notation seamlessly integrates with existing source code repositories so can be operated on by code merging tools. In Umple, model and traditional code hence are not distinguished by such tools. In addition, our approach enables developers to incrementally introduce modeling and/or algorithmic code into their activities.

Writing and editing code or model in Umple can be achieved in any source code editor, although we do provide plug-ins to assist with syntax. Modeling in Umple can be achieved textually by writing the textual notation for modeling elements, or visually by using a UML graphical editor. Changes on either the visual or textual view can be instantaneously reflected on both views, as both visualizations represent the same underlying system.

Umple separates model diagram layout information from the model elements. This information, which can be hidden from the user, is nonetheless maintained as with other aspects of the Umple source code, in the Umple programming language-like syntax. This layout information allows the semantics of a model to be versioned and merged separately from its various diagrams.

3.2 Abstract Textual Notation

The ability to easily version and merge model and code using Umple stems from its abstract textual notation that is managed by the human programmer/modeller, rather than by the *persistence* mechanism of a modeling tool. The Umple syntax is based on a C/Java syntax and so is easily both human readable and editable.

We present two small Umple model segments in the following subsections. For further information on Umple syntax, the reader is referred to (Forward et al, 2009). We have also built an online version of Umple (Lethbridge et al, 2013) that provides an easy forum with examples to demonstrate many of the concepts available in Umple (with a zero install footprint).

In addition to using Umple in collaborative environments to develop software products, we are conducting an empirical study with Umple users using a grounded theory approach to analyze and improve its syntax. Using results of this, details of which are outside of the scope of this paper, we have developed a number of design guidelines for Umple. In particular, its syntax should look similar to and integrate elegantly with the high level programming languages in which it is embedded; it should minimize the use of new keywords, and it should enable users to embed or call native code.

Figure 1 shows a simple UML state machine, where ‘E’ is the event to which S1 responds to; ‘G’ is any Boolean expression, or a Boolean function, or a code segment; and, ‘A’ is a function call, or any arbitrary implementation code.

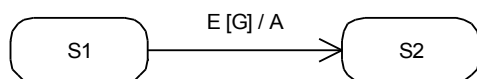


Figure 1: state machine transition

Figure 2 shows Umple code in which two attributes, one association and one state machine are declared. The state machine is the same one that appears in Figure 1. Note how Umple treats modeling abstractions and implementation code uniformly.

In typical modeling tools, when merging of two versions of a model any change in the start state, end state, transition, guard condition, or action may result in a conflict that may require intervention. On the other hand, text-based merging tools like those available within SVN can automatically merge such changes. Even in situations where the changes occur on the same line, a tool like SVN is able to automatically merge the text.

Because the state machine transition is represented in a single line of text (Line 14 in Figure 2), changes to the transition are limited to this line of text, significantly reducing the probabilities and number of conflicts. In addition, when a conflict occurs, it is relatively straightforward to understand what modeling element has changed. In our experience in building systems using Umple, automatic merging has worked very effectively.

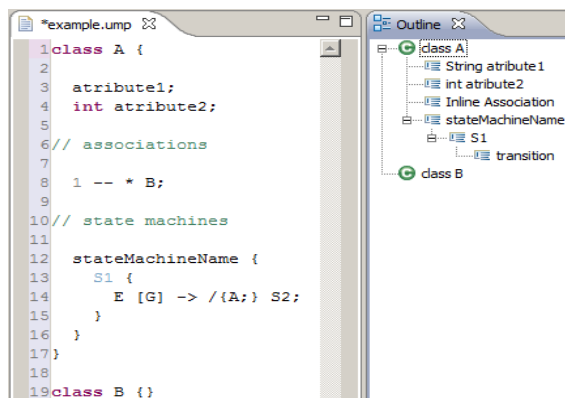


Figure 2: Textual Editor and Outline View

A bi-directional association between class A and class B is illustrated in Figure 2, line 8. Similar to state transitions, the Umple representation of the association is reduced to one line of text. When conflicts occur, it is straightforward to understand what aspects of the model have changed.

Figure 3 shows a section of an airline industry class diagram model. The model can be edited textually or visually, and the layout information is modified and stored in real time as the user manipulates the diagram.

Umple is a full-fledged development platform. The discussion in this paper is limited to its relevance to versioning and merging. Other publications on Umple include (Forward, et al, 2010), (Badreddin et al, 2014), (Badreddin et al, 2014), (Badreddin, 2013), (Badreddin & Lehtbridge, 2013), (Badreddin et al, 2012), (Badreddin & Lethbridge, 2012).

4 ANALYSIS OF VERSIONING AND MERGING USING UMPLE

Text-based merging techniques are widely adopted and familiar to most software developers who collaborate on source code. In this section, we focus our attention on demonstrating model versioning using the SVN diff facility. It is important to note that versioning and merging in Umple handles both code and modeling abstractions residing in the same or separate artifacts uniformly. However, we focus our attention on modeling abstractions. Modellers can choose to inspect the results of merging two versions using a viewer similar to SVN Diff facility (Figure 4), or view the resulting visual models.

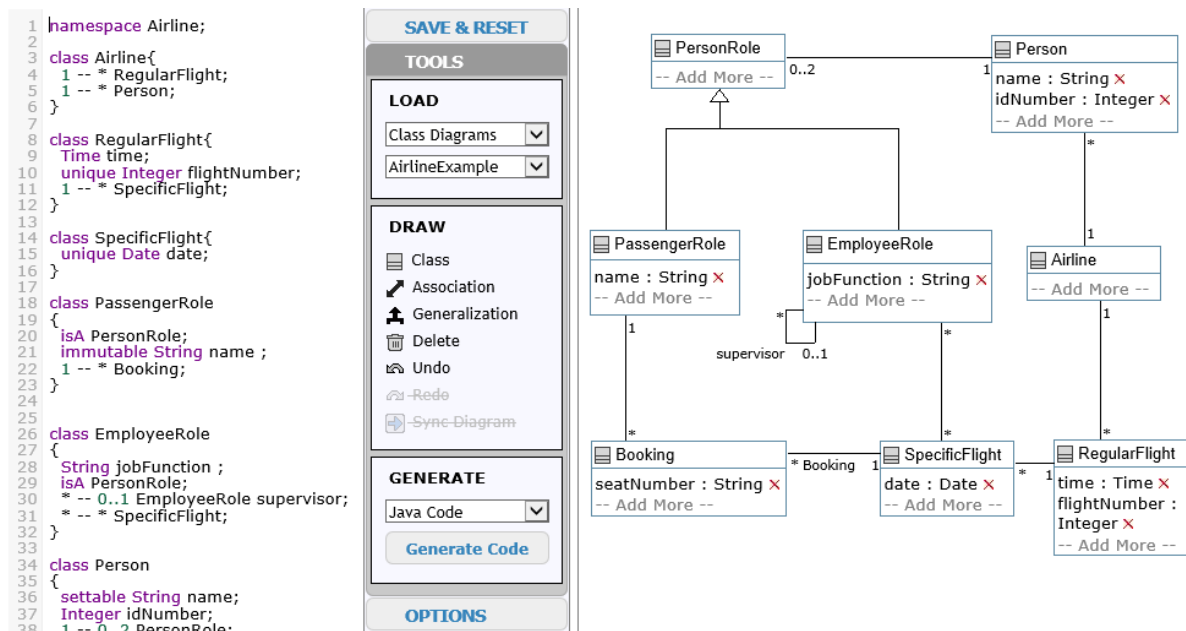


Figure 3: Online visual/textual editor

Figure 4 illustrates merging two versions of the example shown in Figure 2. The following changes to the example were applied and merged with the original model/code:

1. Deletion of *Attribute1*
2. Creation of a New association (to a New class C)
3. Editing of the transition action
4. Transition end state is updated

The right hand side of Figure 4 illustrates the more recent version, while the left hand side illustrates the original (older) version. Updated lines are highlighted and new lines are marked with a plus sign. Similarly, when changes occur only to a portion of a line, the changes are also highlighted. Users can selectively apply any number of changes, without any restrictions on meta-model compliance.

SVN provides a single-pane view and allows for configurations on how to display and deal with conflicts.

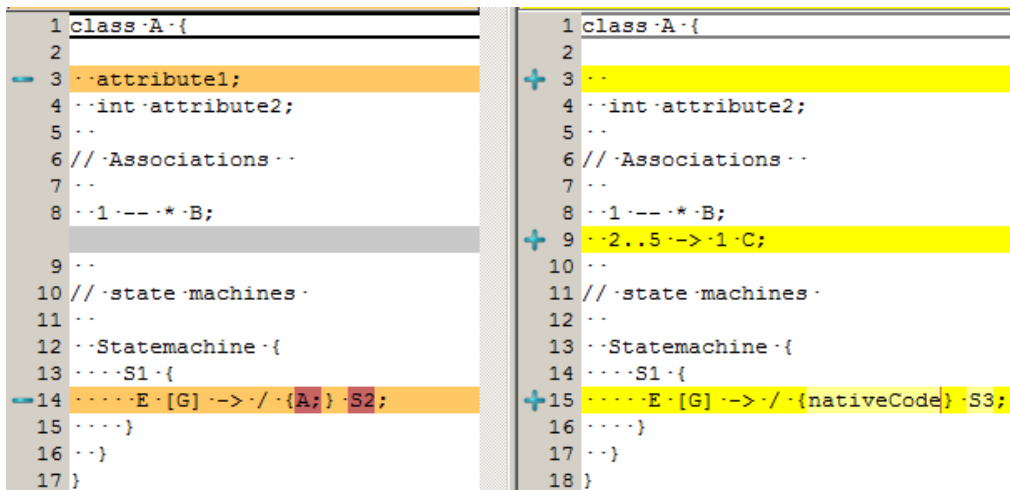


Figure 4: Umlple merging using SVN

If the merge results in an inconsistent model, the violating lines are highlighted in the textual view, similar to any high level programming language editor. Umple's problem view gives the modeler/developer more information on how to resolve the inconsistency. For example, Figure illustrates a scenario where the entry action is not followed by “/”.

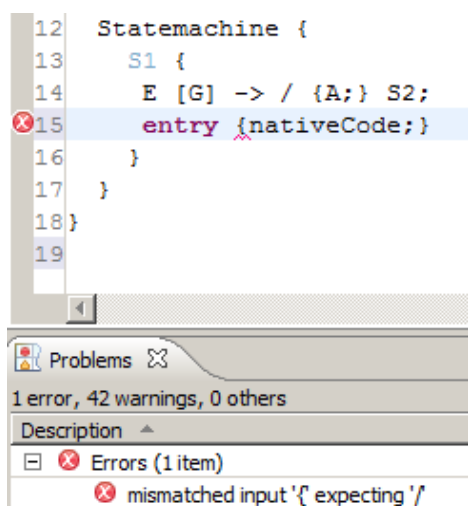


Figure 5: Problems View

4.1 RSA Compare Facility

In RSA, users can compare and merge model versions using a CVS repository, ClearCase (IBM, 2004), or by using the RSA compare facility. ClearCase provides thin clients for remote access, but does not offer additional functionality for model merging. In this section, we illustrate the RSA compare facility.

RSA takes a snapshot of the model with every save. Model structural changes are listed and each can either be accepted or rejected. Structural changes can be addition or deletion of a modeling element. Each structural change is correlated to the merging results, which can be visualized in relation to the model project tree, as in Figure .

The RSA compare facility assumes that the versions always belong to the same model. If two different models are being combined into one, every modification is considered a conflict, which is delegated to the user for resolution. RSA does not allow for temporary meta-model violations, and ignores all layout modifications.

Our choice to compare with RSA is influenced by our judgement that it is the most widely adopted model versioning and merging tool. The RSA

approach is similar to operation-based merging techniques like (Mens, 2000). These approaches rely on the commands performed in the modeling environment to track changes to the model. Such approaches are sometimes referred to as command histories (Berlage & Genau, 1993).

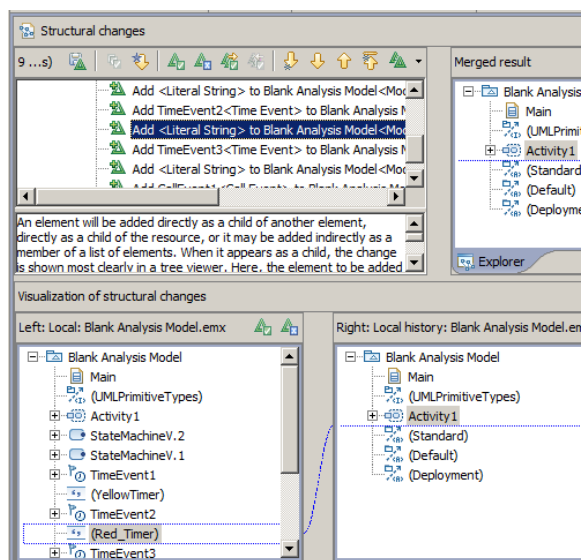


Figure 6: RSA Compare facility

5 UMPLE IN PRACTICE

As discussed previously, the main benefit of our approach is providing uniform merging and versioning for both models and code, because modeling notation can be embedded within code.

Umple has been used to build several systems. The Umple compiler has been fully ported to Umple itself. Umple has been under development since 2007, and all its modeling artifacts have been versioned since inception. We are able to, with minimal storage requirements, review a model revision history (and code) throughout those six years using the same tools developers are familiar with for source code management. There have been over 40 people using and developing Umple, but since Umple can be managed with third-party version control tools the team size can grow indefinitely. In addition, new collaborators require minimal training to start collaborating on models.

In order to minimize the number of merge conflicts, we find it useful to adopt fine-grained revision control (Adams et al, 1986), where we commit changes frequently. Therefore, any change conflict will be small and can be easily managed.

Because the merging is automated, and meta-model violations are allowed, we rely on the facilities provided by the visual and textual editor to resolve any inconsistencies. We combine our model-driven development with a test-driven approach to verify merging sanity. We use a consolidated build script to ensure uniformity between releases and automate the process.

5.1 How Umple Addresses the Barriers to Adoption of Modeling and Merging

In Section 2, we listed some key problems to adoption of modeling tools that relate to difficulties merging and versioning. Here is how using Umple can help resolve these difficulties.

5.1.1 Heavy reliance on subjective conflict resolution

Software professionals have consistently reported high satisfaction with the automated merging of code (Adams et al, 1986). This could be because the nature of code results in minimal overlapping of edits. As we discussed above, using Umple brings these advantages to modeling.

5.1.2 Handling of layout information

Umple has a separate syntax for diagram layout directives (not given in detail in this paper due to lack of space). These directives can be maintained in the same Umple file as the corresponding model, or kept in separate files. Furthermore, their appearance follows the conventions of C-family programming languages just like the rest of Umple, so they can be seen as a harmonious extension to the Umple model-oriented source code.

In Umple, changes purely to layout from one version to another result in deltas that do not impact the core modeling constructs. This is consistent with the emerging practice in modeling tools discussed earlier. Since Umple's layout information is easily readable in textual form, conflicting layout changes during merging can be seen and handled by developers in exactly the same way as conflicting model or code changes.

5.1.3 Adoption of model merging techniques

This is one area where Umple is particularly beneficial. Umple enables modellers to adopt version control without having to change the

existing version-control infrastructure. Developers can therefore benefit from uniform support for evolution of software artifacts throughout the lifecycle of the project (Mens, 2002).

5.1.4 Synchronizing among related MDA artifacts

The changes introduced, whether they are coding, modeling, or layout-related, and whether performed visually or textually, are handled uniformly by the infrastructure. In particular, Umple enables developers to integrate class-diagram modeling constructs, state machine models and methods describing algorithms into the same file should they wish to, or to keep these entirely separate. This provides modellers with a lot of flexibility as they can keep entire systems in as many or few files as they desire. Either way, version control tools' native abilities to manage change sets and merge conflicts solve the artifact-synchronization problem.

Since algorithmic code is embedded in Umple models; the need for versioning the generated artifacts is diminished or eliminated.

5.1.5 Efficiency of versioning and merging

Source code delta algorithms calculate the difference between versions and reduce storage requirements. Modern algorithms reduce I/O operations the time needed to calculate versions compared to historical ones (Hunt et al, 1998). Efficient delta algorithms are typically embedded in merging tools that Umple relies on, like SVN. Scalability is an important feature of any merging technique (Mens, 2002); Using Umple, we are able to efficiently scale to large model sizes with minimal performance implications.

5 ENHANCING CONFLICT RESOLUTION

Umple's textual notation enables developers to merge code and models uniformly. Merging of textual models can be further enhanced if the syntax and semantics of the merging artifacts are taken into consideration.

Syntactic merging (Buffenbarger, 1995), takes the syntax of the software artifact into account. A syntactic merging approach is more powerful than pure textual merging because it can ignore conflicts that are not relevant (for execution purposes) to the

syntax of the language, like comments (other than annotations) and spaces. Merge conflicts can be reported when the merged result is syntactically inconsistent. Examples of such approaches include (Mens, 2000), (Westfechtel, 1991), (Schmidt & Gloetzner, 2008). Coloring techniques, like (Adams, 1986), can enhance the visualization of the changes in the merging process. To apply this to Umple, it would simply be necessary to add awareness of Umple's added modeling constructs to an existing syntactic diff-merge tool.

Semantic merging takes into consideration the semantics of the merged result. Examples of semantic merging include: merging that results in an undeclared variable (static conflict), or merging that results in an inconsistent behavior (behavioral conflict) like (Berzins, 1994) and (Binkley et al, 1995).

Umple merging and versioning tools can be improved by adopting such enhanced textual merging techniques. However, there is an inherent conflict between maintaining language independence with the merging and versioning tools, and introducing syntactic and semantic merging. There are a few approaches that attempt to support semantic merging without compromising their language independence, like (Westfechtel, 1991) and (Edwards, 1997).

6 CONCLUSION

We have shown how Umple, a textual notation that combines modeling elements with traditional algorithmic code, can facilitate merging and versioning of both models and code.

We highlighted our view of the problems of model merging and versioning and how Umple can help solve them. We have reliably used the Umple platform, along with merging and versioning using SVN, to develop Umple itself and other applications.

Key items of future work include exploring the benefits of intelligent merging, as discussed in the last section, and conducting additional empirical evaluation, such as formal usability studies.

Using Umple does not preclude the use of other modeling tools. For example, we have integrated Umple with IBM Rational tools and are working on doing this with open source tools like Papyrus. Such integration enables modellers to use the visual editing capabilities in these tools, and still benefit from the added benefits provided by Umple.

REFERENCES

- Adams, E., Gramlich, W., Muchnick, S. S. and Tirfing, S. "SunPro: Engineering a Practical Program Development Environment," in *An International Workshop on Advanced Programming Environments*, 1986. pp. 86-96.
- Alanen, M. and Porres, I. "Difference and Union of Models". 2003. *Lecture Notes in Computer Science*, Springer. pp. 2-17.
- Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W. and Wimmer, M. "AMOR-Towards Adaptable Model Versioning," in *Proc. of the 1st International Workshop on Model Co-Evolution and Consistency Management*, 2008.
- ArgoUML, "ArgoUML Modeling Tool.", accessed 2009, <http://argouml.tigris.org/>.
- Astah Co. "Astah". 2009.
- Schneider, C. and Zündorf, A. "Experiences in using Optimisitic Locking in Fujaba". 2007. *Softwaretechnik Trends*, vol 27, Citeseer.
- Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Exploring a Model-Oriented and Executable Syntax for UML Attributes." *Software Engineering Research, Management and Applications*. Springer, 2014. 33-53.
- Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity." *Software Engineering Research, Management and Applications*. Springer, 2014. 129-149.
- Badreddin, Omar. "Empirical evaluation of research prototypes at variable stages of maturity", *User Evaluations for Software Engineering Researchers (USER)*, 2013 2nd International Workshop , 10.1109/USER.2013.6603076. 2013 , Pages: 1- 4.
- Badreddin, Omar, Lethbridge, Timothy C., "Model Oriented Programming: Bridging the Code-Model Divide". *ICSE Workshop on Modeling in Software Engineering, 2013, Modeling in Software Engineering (MiSE), 2013 5th International Workshop* , 10.1109/MiSE.2013.6595299. 2013 , Pages: 69 - 75.
- Badreddin, Omar, Andrew Forward, and Timothy C. Lethbridge. "Model oriented programming: an empirical study of comprehension." *2012 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2012.
- Badreddin, Omar. ; Lethbridge, Timothy C. "Combining experiments and grounded theory to evaluate a research prototype: Lessons from the umple model-oriented programming technology", *User Evaluation for Software Engineering Researchers (USER)*, 2012. 10.1109/USER.2012.6226575 , 2012 , Page(s): 1- 4.
- Badreddin, Omar, Timothy C. Lethbridge, and Maged Elassar. "Modeling Practices in Open Source Software." *Open Source Software: Quality Verification*. Springer, 2013. 127-139.

- Berlage, T. and Genau, A. "A Framework for Shared Applications with a Replicated Architecture," *6th Annual ACM Symposium on User Interface Software and Technology*, 1993. pp. 249-257.
- Binkley, D., Horwitz, S. and Repts, T. "Program Integration for Languages with Procedure Calls". 1995. *ACM Transactions on Software Engineering and Methodology*, vol 4, ACM New York, NY, USA. pp. 3-35.
- Berzins, V. "Software Merge: Semantics of Combining Changes to Programs". 1994. *ACM Transactions on Programming Languages and Systems*, vol 16, ACM New York, NY, USA. pp. 1875-1903.
- Buffenbarger, J. "Syntactic Software Merging". 1995. *Lecture Notes in Computer Science*, Springer. pp. 153-153.
- Edwards, W. K. "Flexible Conflict Detection and Management in Collaborative Applications," *10th Annual ACM Symposium on User Interface Software and Technology*, 1997. pp. 139-148.
- Timothy C. Lethbridge, Andrew Forward, Omar Badreddin. "Problems and Opportunities for Model-Centric vs. Code-Centric Development: A Survey of Software Professionals", *C2M:EEMDD 2010*. Available: http://www.esi.es/modelplex/c2m/docum/C2M2010_survey.pdf.
- Forward, A., Badreddin, O. and Lethbridge, T. C. "Umple: Towards Combining Model Driven with Prototype Driven System Development," in *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2010.
- Forward, A., Lethbridge, T. C. and Brestovansky, D. "Improving Program Comprehension by Enhancing Program Constructs: An Analysis of the Umple Language," in *IEEE International Conference on Program Comprehension (ICPC)*, 2009. pp. 311-312.
- Forward, A. "Umple Language Online.", accessed 2012, <http://try.umple.org>.
- Gentleware. "Poseidon for UML". Available: <http://www.gentleware.com>. Accessed 2013.
- Hunt, J. J., Vo, K. P. and Tichy, W. F. "Delta Algorithms: An Empirical Analysis". 1998. *ACM Transactions on Software Engineering and Methodology*, vol 7, ACM New York, NY, USA. pp. 192-214.
- IBM. "IBM Rational Software Architect Modeling Tool", accessed 2009, <http://www-01.ibm.com/software/awdtools/architect/swarchitect/>.
- IBM, "IBM Rational ClearCase". 2004. Rational clearcase.
- Mens, T. "Conditional Graph Rewriting as a Domain-Independent Formalism for Software Evolution". 2000. *Lecture notes in computer science*, Springer. pp. 127-144.
- Mens, T. "A State-of-the-Art Survey on Software Merging". 2002. *IEEE Trans. Software Eng.* pp. 449-462.
- OMG. "UML Diagram Interchange (UMLDI)", accessed 2012, http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML_DI.
- Omondo. "EclipseUML". 2007.
- Pottinger, R. A. and Bernstein, P. A. "Merging Models Based on Given Correspondences," *29th International Conference on very Large Data Bases*, Volume 29, 2003. pp. 873.
- Rho, J. and Wu, C. "An Efficient Version Model of Software Diagrams," *5th Asia-Pacific Software Engineering Conf*, 1998. pp. 2-4.
- Schmidt, M. and Gloetzner, T. "Constructing Difference Tools for Models using the SiDiff Framework". 2008. ACM New York, NY, USA.
- Treude, C., Berlik, S., Wenzel, S. and Kelter, U. "Difference Computation of Large Models," *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007. pp. 295-304.
- Lippe, E. and van Oosterom, N. "Operation-Based Merging". 1992. *ACM SIGSOFT Software Engineering Notes*, vol 17, ACM New York, NY, USA. pp. 78-87.
- Westfechtel, B. "Structure-Oriented Merging of Revisions of Software Documents," *3rd International Workshop on Software Configuration Management*, 1991. pp. 68-79.
- Yang, W. "How to Merge Program Texts". 1994. *J.Syst.Software*, vol 27, Citeseer. pp. 129-135.
- Yang, W., Horwitz, S. and Repts, T. "A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations". 1992. *ACM Transactions on Software Engineering and Methodology*, vol 1, ACM. pp. 354.