Umple: Towards Combining Model Driven with Prototype Driven System Development

Andrew Forward[1], Omar Badreddin[1], Timothy C. Lethbridge[1]

School of Information Technology and Engineering (SITE)
University of Ottawa, K1N 6N5 Canada
aforward@site.uottawa.ca, obadr024@uottawa.ca, tcl@site.uottawa.ca

***Abstract*—The emergence of model driven methodologies is bringing new challenges for software prototyping. Models tend to focus on the design of the system, and are less concerned with, or less able to, support prototype qualities like re-use, evolution, or weaving together independently designed parts. This paper presents a model-oriented prototyping tool called Umple that supports model driven engineering and overcomes some of the challenges related to prototyping in a modeling environment. Umple allows end users to quickly model class and state machine models and to incrementally embed implementation artifacts. At any point in the modeling process, users can quickly generate a fully functional prototype that exposes modeling implications on the user interface, and allows stakeholders to quickly get a feel of how the full system will behave.**

***Index Terms*—Modeling, Prototyping, UML.**

I. Introduction

PROTOTYPING in the software industry plays an indispensible role. "The majority of software projects used some kind of prototype" [1]. In early development activities, a throw-away prototype (also known as a software spike [2]) may be used to help in defining the system scope, assess feasibility,

Manuscript received October 9, 2001. (Write the date on which you submitted your paper for review.) This work was supported in part by the U.S. Department of Commerce under Grant BS123456 (sponsor and financial support acknowledgment goes here). Paper titles should be written in uppercase and lowercase letters, not all uppercase. Avoid writing long formulas with subscripts in the title; short formulas that identify the elements are fine (e.g., "Nd–Fe–B"). Do not write "(Invited)" in the title. Full names of authors are preferred in the author field, but are not required. Put a space between authors' initials.

F. A. Author is with the National Institute of Standards and Technology, Boulder, CO 80305 USA (corresponding author to provide phone: 303-555-5555; fax: 303-555-5555; e-mail: author@ boulder.nist.gov).

S. B. Author, Jr., was with Rice University, Houston, TX 77005 USA. He is now with the Department of Physics, Colorado State University, Fort Collins, CO 80523 USA (e-mail: author@lamar.colostate.edu).

and estimate effort. In later stages, an evolutionary prototype (sometimes called model prototype, or in an agile context a tracer bullet [3]) is typically used whereby the prototype evolves into the working production system.

Each prototype approach has advantages and disadvantages [4]. The decision to develop a specific type of prototype, or not to develop a prototype at all, is based on a number of factors where cost of the prototype is critical [5]. Moreover, it is not unusually for a throwaway prototype to be turned into an evolutionary prototype and later refined into a final deliverable product.

II. Emergence of Model-Driven Practices

Model driven software development, while may still be lower than generally accepted to be a best practice [6], is on the rise. The software industry is increasingly adopting software modeling to generate parts of, or complete, systems. Model-centric software development brings challenges to prototyping:

1) Models are generally more concerned with internal system components and behavior, rather than system user interface, look and feel.

UML, a widely adopted modeling notation, is more concerned with the internal system design and behavior. UML models put less or no emphasis on user interface.

2) Models do not easily evolve into complete systems.

There are significant challenges in model driven development of software. One major challenge is synchronizing generated code with source models, a practice generally referred to as round-tripping [7]. Annotating models with information to guide prototype generation involves the same hurdle.

3) Software modeling notations do not reduce the need for prototyping.

Whether they are throw-away or evolutionary, prototypes are used to brainstorm ideas related to the system, explore different designs, and expose the look and feel of the system early on in the development life cycle. Prototypes are par-

ticularly useful for the business domain stakeholders who typically do not, and need not, understand software modeling notations. Therefore, there is a need to generate tangible artifacts, whether they are screen mock-ups or executable programs. Models, on their own, do not satisfy this need.

4) Software modelers do not fully understand the consequences of their models

Our previous research indicates that modelers quite often create models that do not quite reflect how the system is intended to behave [8].

5) There are limitations in existing approaches to prototype generation from UML models

Existing prototyping approaches rely on one specific modeling notation for code generation. However, in typical software projects, there are more than one modeling notation to describe the same system (e.g. a class diagram, an activity diagram and a state diagram). It is therefore desirable to generate prototypes from a number of related aspects of a model. In addition, annotation of modeling artifacts specifically to support prototype generation is not desirable since models are continuously updated. Unless the annotations are closely integrated into the modeling notations, maintenance of such annotations quickly becomes time consuming.

III. PROTOTYPING

The level of completeness in a prototype is a major factor of success. For example, Jones states "The fact that the prototype was fully functional was critical to its success in eliciting requirements" [9]. However, the cost of developing a prototype and the level of completeness in a prototype are proportional. Therefore, it is important to define what aspects of the functionality of the prototype are most important so that stakeholders can focus on them.

We consider the following properties of prototypes to be crucial:

1) Accurate reflection of system components and data
2) Accurate reflection of current system design directions, and contribution to future design and implementation decisions.
3) Quick and cheap generation of a prototype, since the cost (effort and time) associated with prototyping is a major factor behind the level of completeness (fidelity) of a prototype.
4) Maximize potential of reuse and composition.
5) Support of different levels of abstraction
6) Support of incremental development of features

Throw-away prototypes are cost sensitive. They are an important aspect of exploring a design alternative or new technology, but they should be used sparingly. Stakeholders are generally unwilling to invest any significant amount of effort into throw-away prototypes. Therefore, documentation is even further minimized, increasing the risk of losing the lessons learned.

Evolutionary prototypes evolve into a final working product. Stakeholders are more willing to invest effort and time into evolutionary prototypes as the effort is maintained within the project. Lessons learned are better documented and maintained. This class of prototypes is becoming more attractive for the following reasons:

1) Iterative development methodologies advocate the development of small executable deliverables. These types of methodologies blend well with evolutionary prototype-oriented methodologies.
2) The technology needed to support evolutionary prototyping is beginning to emerge [10]

Because models, especially in iterative development methodologies, are continuously updated and enhanced, it is important to be able to generate prototypes from incomplete models. Generating artifacts from an incomplete model will inevitably require some development intervention to fill in the gaps to produce a tangible artifact that can be used as a prototype. Such effort can be wasted if the underlying tools and development processes do not accommodate iterative development.

IV. UMPLE HIGH LEVEL APPROACH

To support an environment for rapid development of model-centric systems, it is essential to support the development of high system aspects, as well as low level algorithmic specifications at the same time. "Complete prototypes can seldom be produced exclusively with generators and reusable software. Highly complex parts still need to be written by hand" [11]. This is the Umple approach.

Umple combines high level system abstractions and low level specifications in the same development artifacts, making it possible to both design the system, and incrementally add developed components. In addition, Umple combines the modeling of class and state machine models and support prototype generation from both modeling notations.

Umple's complete syntax and grammar is maintained online. We also make available a light version of Umple to illustrate the approach and concept [12].

A. Modeling and Prototyping Approach

Creating models in a typical visual environment can be a demanding task, especially in cases of large models.

Mouse-centric approaches require multiple fine tuning of lines and connections, and continuous dragging and moving of items on the diagram to achieve a nice layout of elements. Textual editors, on the other hand, support a number of features, like syntax-driven editing, that makes the process of creating and editing text less time consuming. In addition, and as we explore in the rest of this paper, textual approaches are better at the representation of reuse concepts like mixins.

Visual models play a better role in communication and collaboration and may be better at conveying information. Umple's goal is to support the benefits of textual environments in creation of models, without compromising the benefits associated with visual models. Umple users can quickly draft models using Umple's high-abstraction notation. Umple blurs the boundaries between modeling and implementation code, and between textual and visual modeling. The Umple prototyping environment allows the user to model textually or visually with changes reflected in both views. More than one modeling notation can be included in the same artifact. In addition, modeling and implementation artifacts are merged in the same file. The next sections illustrate those concepts by simple examples.

B. *Class diagrams*

Figure 1 illustrates the Umple modeling approach for classes and associations. The example illustrates a simple 3-class system.

```
class Student {}

class
CourseSection {}

class Registration
{
  String grade;
  * -- 1 Student;
  * -- 1
CourseSection;
}
```

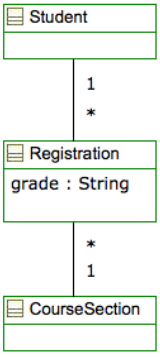

Figure 1: Umple Class Diagram modeling

Umple generates code to implement all combinations of association multiplicities. When a prototype is generated, referential integrity is maintained at run time.

Class attributes contribute to the interface based on their type, and modifier. For example, time and date attributes manifest themselves in the prototype as a drop down menu and a calendar pop-up respectively. Umple also supports "unique" attributes as well as "immutable" attributes, which value cannot change after construction. In addition, the *key* construct defines how the user interface identifies two objects as being equal. Umple users can define what constitutes a key, or use the Umple defaults. Umple key syntax is as follows:

```
Key { attribute1, attribute2, .., ..}
```

C. *State machines*

Umple supports the specification of state machines. Umple supports most of UML 2.2 semantics, including events, signals, guards, transition actions, entry actions, exit actions, nested states and concurrent states. Actions and guards can be written in any implementation language (currently Java or PHP are actively supported).

State machine events by default are generated in a user-interface prototype, discussed later, as buttons. Clicking the button results in invoking that event. The class diagram can have multiple state machines. Therefore, an object can be in one or more states at a time. Each state is represented by the state name in the prototype. The prototype keeps track of the event history invoked on the object to allow for analysis of behavior of the system.

V. UMPLE SUPPORT FOR MIXINS

The concept of the mixin was first popularized in the Ada language [13]. It allows independently developed code to be injected into a set of classes, and thus supports composition of a system. Support for mixins is important for prototype development. They provide modeling and implementation flexibility that is essential for quick and cheap modeling and production of prototypes. We illustrate how Umple mixins enhance reusability of components.

A. *Class diagram mixins*

Umple supports mixins of classes, association, and attributes. An association can be defined within the class that is at either end, or in a separate association entity. If the same class is defined more than once, Umple considers this a single class representation that includes elements from both classes, without repetition. For example:

```
Class A {
  integer x; }

class A {
  Integer Y;
}
```

Umple creates an internal representation of a single class A with both attributes; x and y.

B. *State machine mixins*

Umple supports an unbounded number of state machines in every class, each of which can be defined independently. The same event in Umple can trigger transitions in one or more state machines. Simple functions defining guards and actions can be reused across a number of state machines, or

across classes and components, and again the definitions of these can be defined independently, allowing mixing in of different sets to explore different requirements while prototyping.

The following simple example illustrates a simple traffic control system, where the pedestrian light is dependent on, or controlled by, another state machine controlling the car traffic. For conciseness, we illustrate only partial models.

```
class trafficLightSystem {
  carTraffic {
    Red {
      entry / {goingRed();}
      after(redTimer)[!emergency] -> Yello;
      emergencyNotice -> AllRed;
    }
  pedestrianTraffic
    DontWalk {
      goingRed [!emergency] -> Walk;
      emergencyNotice -> DontWalk;
} }
```

In this example, the event *emergencyNotice* triggers a transition in two separate state machines in the same class. Similarly, the guard *emergency* is used in two transitions in two state machines. The example also shows how an action in one state machine, *goingRed( )*, can function as an event and trigger a transition in an another state machine.

"Reuse of software components in applications still provides some difficulties for a prototyping-based approach" [10]. It is, therefore, our vision to support reuse at various levels of abstraction.

We have so far presented one aspect of reuse and mixin in Umple, where more than one state machine can reuse elements and behave interdependently. We now illustrate another mixin aspect, where complete state machines are reused and customized.

A traffic light's basic operation is timer-based transitions from three states, Red, Green, and Yellow. This simple and basic model can initially be implemented as a stand-alone state machine, and later incorporated into various classes:

For simplicity, we continue to only present partial models.

```
Statemachine coreTrafficController {
  Red {
    After(redTimer) -> Green;
    After(greenTimer) -> Yellow;
    After(yellowTimer) -> Red;
} }
```

In systems where a basic traffic light is desired, the previous standalone statemachine can be referenced as follows :

```
Class TrafficLightController {
  simpleController as coreTrafficController;
}
```

This example creates a state machine called *simpleController* that behaves identically to the *coreTrafficController* state machine.

Some traffic lights may have additional states, like flashing red, or flashing yellow, that are not part of the basic traffic light behavior. Let's call this type of traffic light FrFy for short. Adding such a feature can be accomplished as follows:

```
Class TrafficLightController {
  FrFy as coreTrafficController {
    Red {
    + midnightHour -> FlashingRed; }

    FlashingRed {
     morningHour -> Red;
} }
```

The previous example illustrates a scenario of adding to a basic state machine. The next example illustrates removing and modifying existing elements of a state machine.

Let's assume now we are modeling a traffic light for a high way entrance, and that the light is either Red or Green. We call this traffic light H-way for short.

```
class TrafficLightController {
  H-way as coreTrafficController {
    - After(greenTimer) -> Red;
    ~ After(greenTimer) -> Red ;
} }
```

This example illustrates a scenario where a transition is removed from the model (the first transition in the example above), and an existing transition is modified (the second transition in the example above).

The process of modeling and prototyping controllers may reveal a number of reusable statemachines. Umple textual editor supports a number of features that are geared towards modeling. For example, the outline view allows the modeler to easily locate and select state machines and expose, or hide, their internal elements. Umple support an unbounded number of reusable statemachines.

Such mixin and reuse features, combined with a textual modeling approach, enable modelers to quickly draft, and iteratively refine, a model.

VI. UMPLE PROTOTYPE GENERATOR

At any point in the modeling or development activities, the user can automatically generate a system prototype that allows the user to interact with the system. The prototype generation process is fully automated with development intervention limited to specifying a few preferences. Since defaults are available for the preferences, prototype generation can involve zero effort.

The system prototype has the following features:

1. Allows the user to create instances of classes (objects) and links of associations

2. Classes associations and cardinalities are maintained at run time. For example, an error message is displayed if the user attempts to create an object that results in a violation for the class diagram association.
3. View and update an object attributes.
4. Follow links.
5. Interact with system objects using the exposed public functions.
6. View the history of events and current status of objects with state machines.
7. Change the system's look and feel by applying predefined themes [14].

A. *Umple collaboration support*

The Umple platform uses the textual notation, as opposed to an XMI representation of visual models, for versioning and merging of modeling artifacts. This approach avoids some of the limitation with model-based collaboration. The generated prototypes need not be versioned since they are automatically and entirely generated from Umple models.

VII. UMPLE ARCHITECTURE

Umple's design provides several levels for tool integration. Third party tools can interact with Umple-based systems via XMI. For a more tightly based integration, Umple provides a generator interface to communicate with other model representations such as Umlet, Violet, and Yuml. Finally. Umple provides an API for manipulating a system's model including parsing the Umple language, manipulating the model itself as well as managing the code generator. IBM's RSx, Eclipse and Jazz currently support Umple using their plugin framework. **Error! Reference source not found.** illustrates Umple's architecture.

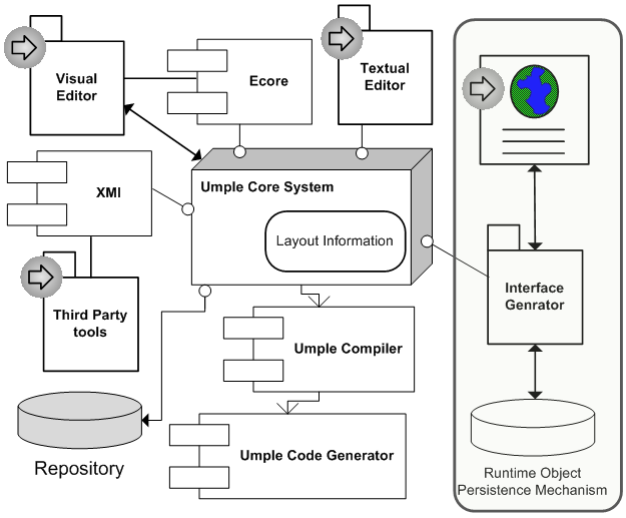

Figure 2: Umple Architecture

Umple's prototyping component is shown to the right of **Error! Reference source not found.**. The interface generator utilizes a runtime Object persistence mechanism to maintain objects created at run time. The interface generator produces html based pages that we host publicly at our web server [12].

Umple users can interact with the system at the 4 interaction points highlighted by arrow inside a circle in **Error! Reference source not found.**. Umple models can be created using the textual or visual editors, or by using any of the supported third party tools. The Umple textual editor provides similar features as a programming language IDE (like Eclipse, or TextMate) including syntax highlighting, context assist, and outline views. The main advantage of the Umple tooling is that is also supports these features for models (not just source code). The visual editor is similar to any modeling software like RSx. We are using Subversion (SVN) repository to provide version control for Umple based systems (including Umple itself), but other Umple based systems have also used GIT. The Umple Compiler and Code Generator are used to generate executables.

VIII. RELATED WORK

There is increasing attention towards the generation of prototypes and user interfaces from UML models. [15] describes an approach where user interfaces can be generated from state machine models that are in turn generated from annotated UML scenario diagrams. Their approach focuses on states and messages in the process of generating prototypes. Similar approaches focus on the dynamics or behavior of the system [statechart and RSA-RT]. Such approaches are generally less concerned with prototyping the

data (instances of classes and their associations) nor with manipulation of the data.

On the other hand, Janssen and Balzert [16, 17] use data structure specifications for UI generation. Li et al[18] generate prototypes from use cases. The use case actions can create or delete an objects; the prototype will check for association validity. These approaches are generally only focused on prototyping the objects and associations of the system.

Umple combines both into one complete prototyping capability: the system's data structure and integrity, as well as the system's dynamic behavior.

A number of prototyping approaches require additional information specific for prototype generation [19]. Such approaches are particularly challenging in model-driven development environments. User interface specific information is generally not managed within the modeling environment. For example, when models evolve, such information needs to be manually added during model transformation. This has significant implications on the cost of developing prototypes. Homrighausen [20] proposed a round-trip prototyping approach where manual changes of the prototype during validation are automatically fed back into the requirement specifications.

While there are a number of textual modeling approaches and tools [21-24], Umple's prototype generation from textual UML models with embedded implementation code is unique. Umple takes the stand of generating prototypes on the fly, as a side effect. The user is not expected to add any specific information to drive the prototype development process. By embedding algorithmic code within Umple models, developers are more likely to maintain and evolve the code as the system matures.

IX. Conclusion

We have presented Umple, a tool that supports rapid modeling and rapid prototyping of systems from UML class and state machine models. The highly abstract textual notation allows Umple users to quickly draft models and utilize the extensive supports of reuse and mixin. Our approach merges models, and code, and generates system prototypes that closely reflect the internal system components and demonstrate system behavior. Our tool merges modeling abstractions and implementation code with the objective of enhancing incremental refinement of models and prototypes.

X. References

[1] H. F. Hofmann and F. Lehner. "Requirements engineering as a success factor in software projects". 2001. IEEE Softwarepp. 58-66.

[2] Memmel, T., Gundelsweiler, F. and Reiterer, H. "Agile human-centered software engineering," in *Proceedings of the 21st British CHI Group Annual Conference on HCI 2007: People and Computers XXI: HCI... but Not as we Know it-Volume 1,* 2007, pp. 167-175.

[3] "The Art in Computer Programming". Standford University, CA: Citeseer, 2001,

[4] "Rapid Prototyping: Principles and Applications". London, U.K: World Scientific Pub Co Inc, 2003,

[5] M. Alavi. "An assessment of the prototyping approach to information systems development". 1984. Communications of the ACM vol 27, pp. 556-563.

[6] Forward, A. and Lethbridge, T. C. "Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals," in *MiSE '08: Proceedings of the 2008 International Workshop on Models in Software Engineering,* 2008, pp. 27-32.

[7] Medvidovic, N., Egyed, A. and Rosenblum, D. S. "Round-trip software engineering using uml: From architecture to design and back," in *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR'99), Toulouse, France,* 1999,

[8] Farah, H. and Lethbridge, T. C. "Temporal exploration of software models: A tool feature to enhance software understanding," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on,* 2007, pp. 41-49.

[9] Jones, M. C., Floyd, I. R. and Twidale, M. B., "Patchwork prototyping with open-source software," in *Handbook of Research on Open Source Software* Anonymous PA, U.S.A: IGI Global; illustrated edition edition, 2007, pp. 126-141.

[10] F. Kordon. "An introduction to rapid system prototyping". 2002. IEEE Trans.Software Eng.pp. 817-821.

[11] G. Pomberger, W. Bischofberger, D. Kolb, W. Pree and H. Schlemm. "Prototyping-Oriented Software Development - Concepts and Tools". 1991. Structured Programming vol 12, pp. 43-60.

[12] "UmpleOnline," accessed 2009, http://cruise.site.uottawa.ca.proxy.bib.uottawa.ca/umpleonline/

[13] E. Seidewitz. "Object-oriented programming with mixins in Ada". 1992. ACM SIGAda Ada Letters vol 12, pp. 76-90.

[14] Solano, J. "Exploring how model oriented programming can be extended to the UI level". Available: http://www.site.uottawa.ca/~tcl/gradtheses/jsolano/

[15] M. Elkoutbi, I. Khriss and R. K. Keller. "Automated prototyping of user interfaces based on uml scenarios". 2006. Autom.Software.Eng vol 13, pp. 5-40.

[16] Janssen, C., Weisbecker, A. and Ziegler, J. "Generating user interfaces from data models and dialogue net specifications," in *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*, 1993, pp. 418-423.

[17] H. Balzert. "From OOA to GUIs: The janus system". 1996. JOOP vol 8, pp. 43-47.

[18] X. Li, Z. Liu, J. He and Q. Long, "Generating a Prototype from a UML Model of System Requirements," Springer, Tech. Rep. UNU-IIST Report No. 307, 2004.

[19] M. Elkoutbi and R. K. Keller. "User interface prototyping based on UML scenarios and high-level Petri nets". 2000. Lecture notes in computer sciencepp. 166-186.

[20] A. Homrighausen, H. W. Six and M. Winter. "Round-trip prototyping based on integrated functional and user interface requirements specifications". 2002. Requirements Engineering vol 7, pp. 34-45.

[21] avishn. (2009, "ModSL - text-to-diagram UML sketching tool". vol. 0.6.4, 2009.

[22] A. Fadila and G. Said. "A new textual description for software process modeling". 2006. Information Technology Journal vol 5, pp. 1146-1148.

[23] G. Fliedl, C. Kop and H. C. Mayr. "From textual scenarios to a conceptual schema". 2005. Data Knowl Eng vol 55, pp. 20-37.

[24] "Human-Usable Textual Notation," accessed 2010, http://www.omg.org/technology/documents/formal/hutn.htm