Constraint Animation Using an Object-Oriented Declarative Language

Jeff Gray and Stephen Schach
Department of Computer Science
Vanderbilt University
Nashville, TN
{jgray, srs}@vuse.vanderbilt.edu

Abstract- Prototypes can be an effective way of interacting with an end-user to validate that the user's requirements have been correctly captured. In the formal methods community, specification animation has been investigated as a way of creating a kind of prototype that is generated from a formal specification. Enriching UML diagrams with OCL constraints can provide the formality that is needed to animate the diagrams without the need for a more rigorous formal specification language. This paper provides an overview of issues concerning specification animation and describes an initial attempt at an animation environment for UML/OCL. We translate the UML/OCL into an object-oriented declarative language, Prolog++, and utilize a primitive animation environment that allows both a developer and client to explore the validity of the specification. In particular, in this paper we focus on animating the effect of constraints.

1 Introduction

Language exists to communicate whatever it can communicate. Some things it communicates so badly that we never attempt to communicate them by words if any other medium is available

C.S. Lewis [33]

According to the *Oxford English Dictionary*, the 500 words used most in the English language each have an average of 23 different meanings. The word "set," for instance, has 430 distinctly different senses. Over 25 pages and approximately 60,000 words were required to define this word [40]. The variance of word meanings in natural language has always posed problems for those who attempt to construct an unambiguous and consistent statement. It is often the case that a written statement could be interpreted in several ways by different individuals, thus rendering the statement subjective rather than objective.

Best Conference Paper Award:
Proceedings of the 38th Annual ACM SE Conference, Clemson, SC, April 7-8, 2000, pp. 1-10.

One of the first detailed examinations of this problem with respect to the specifications of computer systems is contained in [25]. Hill provides numerous examples to illustrate this common problem. Additionally, Peter G. Neumann represented this point by constructing a sentence that contained the restrictive qualifier "only." He then showed that by placing the word "only" in 15 different places in the sentence resulted in over 20 different interpretations [37]. Moreover, other words like "never," "should," "nothing," and "usually" are sometimes applied in a manner in which a double meaning can be ascribed. We often rely on metaphors in our daily communication. As noted by Boyd, "Metaphors play a fundamental role in communication. Observe that natural language is rich with metaphors. Our words are pregnant with meaning" [5].

Occasionally the ambiguity found in natural language may evoke images of the ridiculous while at other times it may be the source of humor.[1] For example, suppose a friend said to you, "I found a smoldering cigarette left by a horse." Two meanings could be given to this statement: 1) a person found a smoldering cigarette near a horse, or 2) a horse dropped a cigarette that you later found. Of course, the most likely meaning is given in the first explanation, but the example highlights a problem that can occur when the immediate context and personal experience do not necessarily suggest the most probable meaning.

A famous case study in which the reliance of natural language resulted in numerous specification errors is documented in [34].[2] Originally, Peter Naur specified a line-editing problem using English [36]. Several authors, most notably [13], used the problem to illustrate how their specific technique could be used to uncover errors. Meyer's detailed formal analysis of the problem revealed that successive examinations and attempts at respecification using natural language, including the testing techniques given in [13], continued to introduce additional errors.

[1] A list of ambiguous and inconsistent statements is being compiled and can be found at:
http://www.vuse.vanderbilt.edu/~jgray/ambig.html

[2] See [42] for a detailed chronicle of the problem.

This example points to the potential confusion that can result when using natural language. That is, informal descriptions are subject to the vagaries and ambiguities of the natural language in which they are expressed. If simple sentences are vulnerable to ambiguity, one can only imagine the potential problems that exist within a software requirements specification written entirely in natural language. Such documents can easily be hundreds or thousands of pages long. The possibility of ambiguities and inconsistent statements existing in such documents is very real. As described in [10], International Space Station project developers rely heavily on natural language specifications. They present a sample of the type of ambiguous natural language existing in the Space Station specification and note that their attempts at converting it into a formal tabular notation resulted in four different interpretations by four different teams. They state that, "the task of generating formal specifications from this documentation is fraught with difficulty" [10].

The Unified Modeling Language (UML) is the industry-standard notation for object-oriented analysis and design [3]. UML diagrams can be supplemented with constraints specified in the Object Constraint Language (OCL) [49]. The same problems noted above could occur in a UML diagram that relies extensively on natural language to explain the meaning of things that are not explicit in the diagram. As Stuart Kent writes, "there are some constraints that can not be expressed diagrammatically using existing modeling notations" [28]. Enriching a UML diagram with OCL constraints can help in reducing the ambiguity [39], [48]. Proponents of OCL have asserted that, "Constraints make models less ambiguous" [49]. OCL can be used to describe the pre/post conditions of methods, as well as provide a standard way of expressing the meaning of constraints between objects in a way that is not subject to multiple interpretations.

In past years, we were interested in methods for improving the formalization of OOAD models [17]. We focused our efforts on methods for translating the UML into formal object-oriented specification languages like Object-Z [7], [45] and [31]. Many researchers have made significant contributions in this area; see [4], [6], and "The precise UML Group," pUML (http://www.cs.york.ac.uk/puml/).

A second benefit can be realized from a disciplined use of OCL. As the level of formality and preciseness increases in a UML diagram, the ability to execute or animate the diagram also increases. This allows for a kind of prototype that is constructed from the semantics given to the combination of the diagram and the OCL expressions. This paper describes our initial efforts at creating an animation environment for UML/OCL.

In Section 2 of this paper we provide an introduction to the specification animation literature. That section is followed by a discussion in Section 3 of the current approach that we use to animate UML/OCL by translating the diagrams into a declarative object-oriented programming language. A small example is then presented in Section 4 to illustrate the approach. Section 5 describes an animation environment that we have created. A summary section concludes the paper.

2 Specification Animation

A stander-by may sometimes, perhaps, see more of the game than he that plays it.

Jonathan Swift [8]

Whether it be a contractor or architect interacting with a future building owner, or a developer meeting with a client, prototyping is often used as a means of arriving at consensus with regard to the customer's requirements [1]. For example, an analysis of 39 case studies that used rapid prototyping was conducted by Gordon and Bieman [15]. Their work is a report on a number of commercial software projects in which prototyping was used with some success. Of these, 33 were considered successes, 3 were considered failures, and 3 were not rated.

In [27], results are presented from a study which examined the efficacy of 15 different customer-developer links. A customer-developer link is defined as the facilities used to exchange information between the client and the developer. In conducting their study, Keil and Carmel visited 17 software development companies and collected information about 31 different projects. The development environments used to classify each project were *package* (commercially "shrink-wrapped" software) or *custom* (for contract or in-house software) environments. Of the 31 projects, only 14 were determined to be "successful" (a questionably subjective term which was interpreted by each individual project manager). The results of this study lend support to the technique of rapid prototyping. For custom projects, the user-interface/requirements prototyping links were used the most on successful projects and were ranked as 2nd (user-interface prototyping) and 3rd (requirements prototyping) in terms of overall effectiveness. The results for packaged software were not quite as supportive (user-interface was 3rd and requirements was 5th in terms of effectiveness) [27].

In the formal specification literature, animation refers to a kind of prototype resulting from an executable specification. With animation, a specifier may pose queries against the specification that can be answered in an automated way. An advantage of this approach is obtained when both the specifier and client come together to validate the

specification. This scenario provides an environment to ask 'what-if' questions without the explicit construction of a prototype. Rapid feedback from the animation session can help to ensure that the specification actually represents what the client or user intended.

An additional advantage is that the animation approach makes the underlying notation and formalism transparent. The user of the animator does not need to have a thorough understanding of the notation used to represent the system. Leaders in the formal methods community have written that it is a myth to state that formal methods require highly trained mathematicians [18]. Although it may be true that one does not need a degree in mathematics to specify a system formally, it is also true that many beginners do indeed have a difficult time in reading and writing formal specifications. It is a skill that requires practice and patience. An initial investigation into the difficulty of applying discrete mathematics in formal specification is described in [11]. In this experiment, 62 computer science graduate and undergraduate students were asked to read a small specification written in the popular Z specification language [44]. Each student was then asked three questions. An interesting result was that nearly a third of the students could not answer a single question correctly about the specification. Finney wrote, "…in general the students found it difficult to understand *any* of the very simple Z specifications" [11].

One could argue that the same observations are true for UML diagrams that are enriched with OCL. A customer will most likely not understand all of the significant analysis/design knowledge embodied in the diagrams and constraints. We believe that benefits can be realized when animation is used to make the diagrams and constraints transparent to the end-user who wishes to query the specification.

2.1 Opposition to the Approach

There have been several arguments put forth against animation. The main objection in [21] concerns the tradeoff between the expressiveness of a formal notation and its executability; that is, clarity is to be given a higher priority than executability. Any attempt at animation is seen as a minimization of this priority. An example of this tradeoff can be found in [26], where the greatest common divisor function was specified. A concise and clear specification of the `gcd` function is terribly inefficient in terms of its executablity. A refinement of the function is shown to improve executability, but the refinement is more difficult to understand.

A concise specification that contains negation and non-determinism can lose some of its clarity when it is refined into a form that is more amenable to animation. To achieve animation, there is also a tendency to over specify a problem while introducing too much implementation detail. Hayes and Jones also argue that formal specifications are to be read and that other techniques for validation should be employed as an alternative to animation.

A response to these claims can be found in [12]. Fuchs shows that the examples in the Hayes and Jones paper, "can be directly translated into executable form on almost the same level of abstraction, and without essentially altering their structure." Much of that paper is focused on demonstrating how to transform the Hayes and Jones examples into an executable form that does not radically lower the abstraction level. Positive arguments for the use of animation are also made in [16], where animation is used to validate a commercial software package.

2.2 Related Work

There are some obvious observations that can be made from the literature on animating formal specification languages:

1. Most of the research has focused on the Z specification language [44], or subsets of Z. Examples of work that focused on Z are the Surrey Z Animator (SuZAn) [30], the animator described in [50], the work of Goodman [14], ZAL [43], the work of Utting [47], Z Animation System (ZANS) [51], Prolog Z Animator (PiZA) [24], and Possum [23]. Similarly, research has also been conducted in animating Object-Z [22].
2. Many of the animators enumerated above use a declarative programming language to implement the underlying functionality of the animator. The most represented language is Prolog, while a few animators have used Haskell.
3. Most of these animation systems, however, require the user either to understand portions of the formal specification, or to have familiarity with the programming language that is used to implement the animator.

In the context of UML, there is some similarity in our research interests with the description of tools offered in the commercial product from IFAD (VDMTools); see http://www.ifad.dk. The IFAD tool suite provides a link with Rational Rose to translate UML diagrams into VDM or VDM++ [9]. Another tool in the suite will interpret and debug VDM/VDM++ specifications. This commercial tool, however, deviates from our goal of making the underlying formalism transparent to the end-user.

3 Animating UML/OCL with Prolog++

Specifications, by their very definition, are declarative in nature; that is, they specify *what*, rather than *how*, the system behaves. OCL is a declarative language [49]. Declarative logic programming languages that employ a Horn clause syntax, like Prolog, have been beneficial to animator construction, as noted in Section 2.2. For our research we have utilized Prolog++, a Prolog environment from Logic Programming Associates (LPA) that offers object-oriented extensions to a standard Prolog compiler (see http://www.lpa.co.uk). We chose this particular language because of its declarative nature and the more natural mapping from UML, due to its object-oriented additions. In standard Prolog, a program is created by defining various logical predicates. In Prolog++, those predicates are moved into a class structure and serve as methods of a class.

Like most object-oriented languages, Prolog++ offers syntax and semantics for representing class and instance attributes, as well as class and instance methods. Visibility control of attributes and methods are available, although there is no current support for `protected` visibility; only `public` and `private` visibility control is allowed. Multiple inheritance is supported, as well as syntax for explicitly representing the aggregation that may occur within a class. Please refer to [35] for a detailed description of this language.

The main task in constructing an animator is the formulation of a mapping from the conceptual model to the declarative programming language that serves as the core engine of the animator. A major sub-task of creating this mapping is the construction of a library of routines in the underlying language that represent the semantics of operations used in the modeling language. For example, OCL provides several operations that are often used to specify invariants on subclasses (e.g., *oclIsKindOf*, *oclIsTypeOf*, and *oclType*). Because Prolog++ does not provide constructs for supporting these operations, we needed to supplement our class definitions with reflective type information. We accomplished this by introducing a class attribute, named `class_type`, into each Prolog++ class definition. The subclass operations are then defined with respect to this attribute. As an example, the following definition of the `oclIsTypeOf` operation exists in one of the *mixin* classes[3] that we have created to emulate these operations in Prolog++:

```
oclIsTypeOf(AType) :- self@class_type = AType.
```

[3] All base classes must inherit from these *mixin* class.

Collections are used extensively in OCL. Prolog++ offers a useful construct that provides access to all instances of a particular class. This can be most helpful in the animation of collections composed of instances. In Prolog++, all instances of a particular class can be sent a message with the following command:

```
(all instance class)<-someMessage.
```

Using variations of this construct as a template, many of the collection-based operations of OCL (e.g., `forall`, `exists`, `collect`, and `select`) can be translated into Prolog++. Obviously, this construct is helpful when mapping the OCL `allInstances` operation.

In addition, OCL collections have predefined properties. The Prolog++ `all instance` template can be used to implement these properties as well. A collection property that is often referenced is the `size` property, which returns the number of elements that are contained in a particular collection. We have defined a Prolog++ implementation of `size` in a *mixin* class. It is implemented by having each element of a collection increment a class attribute that contains the overall size. The `size` method can also be used to provide a facility for verifying the multiplicity constraints from a UML class diagram. This is accomplished by applying the `size` method to the collection resulting from navigating an association. The `isEmpty` property can be used to determine if a particular collection contains any elements. This can be naively translated as a special case where `size` is greater than zero. Please see Section 4.3 for an example that applies the `size` method.

If a collection is composed of numeric elements, the `sum` property can be meaningfully applied. This property represents the sum of all elements in a collection. The same technique that is used for implementing the `size` property can be used to implement `sum`; that is, using a global class attribute to temporarily serve as a placeholder for the summation when an iteration over a collection is performed.

By defining the Prolog++ equivalent of OCL primitive operations and properties in *mixin* classes, we can more easily map the specific OCL constraints into a form that can be executed from within the Prolog++ environment.

4 Example

In this section, we demonstrate the approach of validating association constraints. We use an example that models the basic components of an audio system; see Figure 1. The model example is taken from [38].

In this model, a connection is made between components through a *port*. Some components, like power amplifiers and preamplifiers, have both an input port and output port. However, microphones have a single output port, whereas speakers have only input ports. The src and dst roles of the Connection association model the coupling between audio components. Impedance attributes have been added to OutputPort and InputPort, but the diagram has been kept simple to highlight the associations that will be under consideration.

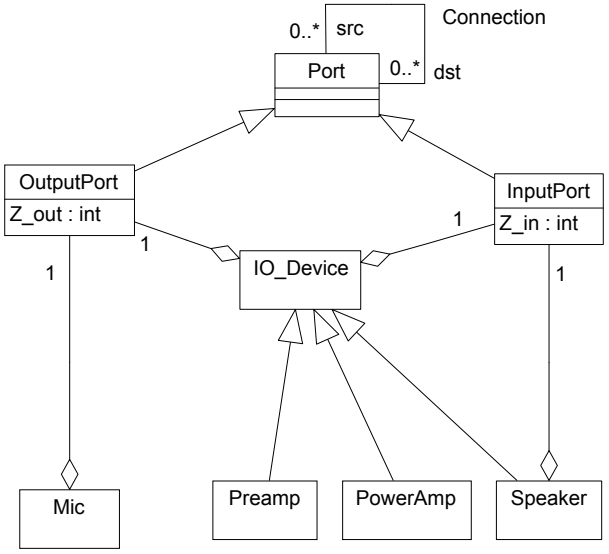

Figure 1. Audio System Example

A problem with the model is that it allows full connectivity between any objects that are instances of Port or Port's subclasses. For example, the above model would allow an input port to connect to itself - an illegal connection in this domain. OCL can be used to enrich the UML class diagram by describing rules about valid connections. Without OCL, such rules would have to be described in natural language. OCL allows a more precise specification of the connection rules. Using OCL also opens up the possibility of translating the constraint definition into an animation environment or programming language.

There are three types of rules that we will specify using OCL:

- valid connection of port types (e.g., input ports can connect only to output ports),
- valid connections between audio components (e.g., microphones and preamplifiers),
- the necessary existence of certain components (e.g., at least one power amplifier is needed).

4.1 Specifying connection types

Our model should specify that valid connections exist when an input port is connected to an output port. However, it is invalid to connect an input port to another input port, or an output port to another output port. This can be specified in OCL as:

```
Connection->forAll(c |
                   c.src.oclIsTypeOf(OutputPort)
                and c.dst.oclIsTypeOf(InputPort))
```

To represent this constraint in Prolog++, the class_type attribute must be inserted into each class, as discussed in Section 3. Therefore, the outputPort class would contain a class attribute definition like the following:

```
public class attributes
  class_type = 'outputPort'.
```

A similar declaration would need to be defined in the inputPort class as:

```
public class attributes
  class_type = 'inputPort'.
```

In the Prolog++ `port` class, the `inv_Connection` method can then be used to represent the semantics of the above OCL constraint. This method can be written more concisely using the `oclIsTypeOf` method described in Section 3:

```
inv_Connection :-
   (all instance class)<-
          (@src<-oclIsTypeOf('outputPort'),
           @dst<-oclIsTypeOf('inputPort')).
```

4.2 Constraining relationship participation

We also need to specify the valid connections that can occur between the various audio components. For example, our model will specify that a microphone can connect to only a preamplifier; the output of a preamplifier can connect to only a power amplifier; the output of a power amplifier can connect to only a speaker. The OCL equivalent of these constraints could be written as:

```
Mic->forAll(m | m.outputPort.dst ->
            forAll(i | i.preamp))

Preamp->forAll(p | p.outputPort.dst ->
               forAll(i | i.poweramp))

PowerAmp->forAll(a | a.outputPort.dst ->
                 forAll(i | i.speaker))
```

Alternatively, the above expression that constrains a microphone's output port could be specificed as:

```
Mic->forAll(m |
            m.outputPort.dst.oclIsTypeOf(PreAmp))
```

Using an approach similar to Section 4.1, the above OCL constraint can be translated into Prolog++ by navigating from the output port part and checking that the class type of its destination port is a preamplifier. The Prolog++ for this constraint would be translated as:

```
inv_Mic_Dst :-
    (all instance)<- (outputPort@dst
                      <-oclIsTypeOf('preAmp')).
```

4.3 Specifying component existence

A final constraint that we want to specify concerns the requirement that at least one power amplifier must exist in our audio system. In OCL we can specify this as:

```
PowerAmp.allInstances->size >= 1
```

As described in the Section 3, each base class in a hierarchy must inherit from a *mixin* class that contains a method for computing the size of a collection. The *mixin* class also has a class attribute (`size_att`) that is used to compute the size. The `size` method can be written in Prolog++ as:

```
size_func :- size_att += 1.

size(S) :- size_att := 0,
           (all instance class) <- size_func,
            S = @size_att.
```

With the above primitives in place, the actual constraint can be written simply as:

```
inv_powerAmp_existence :- size(S), S >= 1.
```

After the construction of basic primitives in *mixin* classes, OCL constraints can be concisely represented using the declarative power of Prolog++.

5 An Animation Environment

You need to be able to execute, debug, and simulate your model if you want to have any kind of feedback of what the system will actually do

David Harel [19]

A major goal in this work is to make the underlying Prolog++ transparent to the user. During a prototype demonstration, there is little advantage in revealing the underlying source code to the customer. Likewise, specification animation should not involve the customer in the details of Prolog++. However, it may be necessary in some situations for a developer to access the underlying representation during an animation, but not the customer. We provide access to the base Prolog++ environment, but the focus from the customer's aspect should be a more suitable view of the results of an animation.

Figure 2 illustrates the key components of our initial UML/OCL animation environment. Developers and customers interact with each other as they validate the UML/OCL models while using the animation environment. A tool provided by Logic Programming Associates, the Intelligence Server, allows programs written in other languages to access the Prolog++ environment. In our animator, this is a valuable asset because it allows the creation of the animation environment using a language[4] that has a powerful user-interface focus while treating the Prolog++ environment as our fundamental execution engine for animation. The animation environment constructs Prolog++ queries and issues them to the Intelligence Server. As Prolog++ executes the queries, results are returned back to the environment from the Intelligence Server.

To achieve transparency, a hierarchical view of the classes and existing objects is available. From this view, a user of the animator may create and delete objects, view and assign values to object attributes (if set/get methods are defined), and check constraint invariants; see Figure 3. The animator interacts with the Intelligence Server and issues the appropriate queries to process the user's commands and update the hierarchical view. When needed, a developer may also issue Prolog++ queries directly and alter the state of the animation. The developer can see the effect of a query in a special window that captures the result returned from the Intelligence Server.

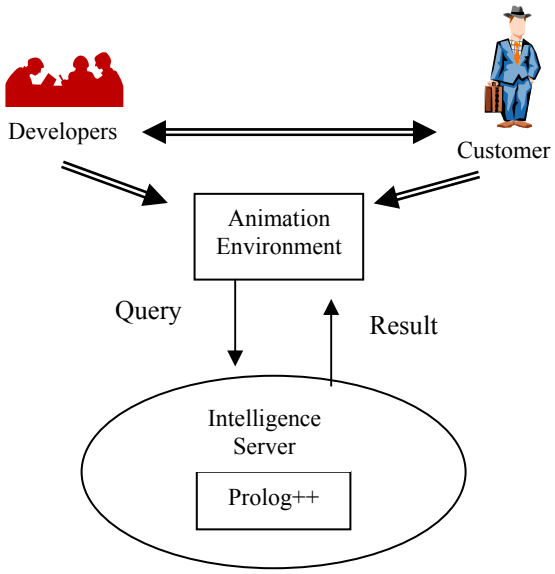

Figure 2. Components of an Animation Environment

[4] Our animation environment has been constructed using Inprise's (Borland) Delphi.

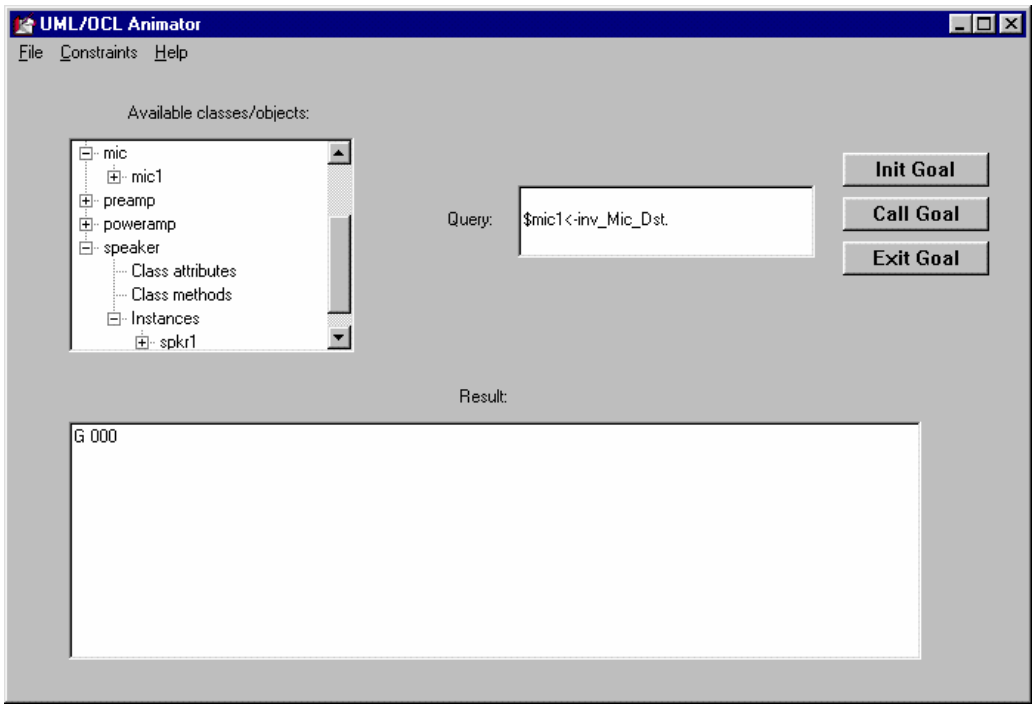

Figure3. A Prolog++ Animation Environment

As with other animators, our animator allows a user to close an existing animation and load a new model description. Additionally, a user can save the state of the current animation session. In this case, all of the objects that exist during a session are serialized to an output file and the state can be loaded back into the animator during a future session.

Through the animator, the result of constraint invariants can be viewed. Constraints can be invoked at either the object or class level. At the class level, the constraints of all objects of a particular class are checked. Also, an option is provided to check every constraint in the model.

6 Conclusion

This returns us to the underlying tension in the software process: that between the subjective and the objective, between the holistic mental view and the rigorous formal model. The first describes what is needed; the second ensures that what was requested is delivered.

Bruce Blum [2]

An often cited adage states that, "A picture is worth a thousand words." In most cases, this is meant to signify a positive trait that expresses the subjective power of a visual stimulus. For example, a piece of art often embodies an observable image in addition to eliciting emotional and visceral responses. Sometimes the perceptions of the viewer extend any original intention that the artist attempted to render; this is the subjective beauty of art. In a sense, this adage focuses on the *connotative* expressiveness of visualization.

However, what might be beneficial from a subjective viewpoint could prove detrimental when attempting objectivity. Using this adage, a problem may occur when a visualization does indeed represent a thousand words; especially when only about 50 of those words were intended! An illustration representing this view is presented in [41] and is accompanied by a statement that summarizes the utility of graphical representations, "In considering representations for programming, the concern is formalisms, not art – precision, not breadth of interpretation."

The designers of UML have worked diligently on the difficult task of documenting the semantics of the UML diagrams [3]. The annual OOPSLA workshop on Behavioral Semantics [29] and the work of the pUML have also concentrated efforts in this important area. The criticality of these efforts can be summarized by the following statement:

However, many methodologists fail to rigorously define the semantics of the languages. Without a rigorous semantic definition, precise model behavior over time is not well defined and full executability and automatic code synthesis is impossible.

Harel and Gery [20]

There are some things that cannot be specified with fine-grained detail in a UML diagram. Often, those things that cannot be specified precisely in a visual diagram are specified in natural language as commentary that is peripheral to the diagram. In a previous section of this paper we have argued against the danger of ambiguity that can result from a reliance on natural language.

The OCL has been proposed as an alternative to natural language in the specification of those things that cannot be visually described [39], [48]. We believe that the enrichment of UML diagrams with OCL not only increases the precision of a UML diagram, but also presents opportunities for animating those diagrams using a declarative language. Specification animation allows a developer to interact with the customer in a validation context while hiding many of the underlying diagrams and formalisms from the client.

Our initial work has focused on discovering techniques for mapping parts of the UML and OCL to an object-oriented declarative programming language. In particular, we have focused on the translation of invariant constraints from OCL to Prolog++ and integrating the translation into an environment that allows a developer to interact with the client. We plan to improve on this initial work in several areas.

Our most immediate plan for future work is the incorporation of method pre/post-conditions into the animation. Modeling the behavior of a method through pre/post-conditions is a most important aspect toward providing a meaningful animation. We will also investigate the roles that other UML diagrams (e.g., state diagrams) play in an animation session.

The Prolog++ code that we currently load into our animation environment and feed through the Intelligence Server is manually translated from UML/OCL. An obvious feature that we would like to add is a technique for automatically translating the UML/OCL into Prolog++. We have already constructed an OCL parser for a project that is unrelated to this work. Our research colleagues at Vanderbilt's Institute for Software Integrated Systems (ISIS)[5] have performed work on Model Integrated Computing (MIC) [46] that has resulted in a Graphical Modeling Environment (GME) for creating, among other things, UML diagrams [32]. We would like to explore the possibility of using environments like the GME, or Rational Rose and the Rose Extensibility Interface (REI), to extract a model and automate the translation.

[5] http://www.isis.vanderbilt.edu

References

1. Berry, Daniel M., "Software and House Requirements Engineering: Lessons Learned in Combating Requirements Creep," *Requirements Engineering Journal*, 3:3&4, 1999, pp. 242–244.
2. Blum, Bruce, "A Taxonomy of Software Development Methods," *Communications of the ACM*, November 1994, pp. 82–94.
3. Booch, Grady, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley. Reading, Ma, 1999.
4. Bourdeau, Robert H., and Betty C. Cheng, "A Formal Semantics for Object Model Diagrams," *IEEE Transactions on Software Engineering*, October 1995, pp. 799-821.
5. Boyd, Nik, "Using Natural Language in Software Development," *The Journal of Object-Oriented Programming*, February 1999, pp. 45–55.
6. Bruel, J.M., R.B. France, and E.B. Ferndandez, "Formal Specification of a Multimedia Conferencing Authorization System," *Proceedings of the Fifth OOPSLA Workshop on Specification of Behavioral Semantics*, Haim Kilov and V.J. Harvey, Eds., San Jose, CA, October 1996, pp. 19–32.
7. Carrington, David, David Duke, Roger Duke, Paul King, Gordon Rose, and Graeme Smith, "Object-Z: An Object-Oriented Extension to Z," *Formal Description Techniques II* (1989), North-Holland, 1990, pp. 281–296.
8. Davis, Herbert, Ed., "A Critical Essay Upon the Faculties of the Mind," in *Jonathan Swift - Tale of a Tub: With Other Early Works*, Basil Blackwell & Mott Ltd., Oxford, UK, 1965.
9. D✧rr, E.H., and N. Plat, "VDM++ Language Reference Manual," Afrodite (ESPRIT-III project number 6500) document AFRO/CG/ED/LRM/V10, Cap Volmac, August 1995, pp. 1–85.
10. Easterbrook, Steve M., and John Callahan, "Formal Methods for V&V of Partial Specifications: An Experience Report," *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, Annapolis, Maryland, January 5-8, 1997.
11. Finney, Kate, "Mathematical Notation in Formal Specification: Too Difficult for the Masses?" *IEEE Transactions on Software Engineering*, February 1996, pp. 158–159.
12. Fuchs, Norbert, "Specifications are (Preferably) Executable", *IEE Software Engineering Journal*, September 1992, pp. 323–334.
13. Goodenough, John, and Susan Gerhart, "Towards a Theory of Test Data Selection," *IEEE Transactions on Software Engineering*, June 1975, pp. 156–173.
14. Goodman, Howard S., "Animating Z Specifications in Haskell Using a Monad," School of Computer Science,

University of Birmingham, England, November 1993, pp. 1–28.
15. Gordon, V.S., and J.M. Bieman, "Rapid Prototype Lessons Learned," *IEEE Software*, January 1995, pp. 85–95.
16. Gravell, Andrew and Peter Henderson, "Executing Formal Specifications Need Not Be Harmful," *IEE Software Journal*, March 1996, pp. 104–110.
17. Gray, Jeff, "Improving Completeness and Consistency in Object-Oriented Analysis through Adaptable Formalisms," *ACM Mid-Southeast Conference*, Gatlinburg, TN, November 1995.
18. Hall, Anthony, "Seven Myths of Formal Methods," *IEEE Software*, September 1990, pp. 11–19.
19. Harel, David, "On Modeling and analyzing Object Behavior," Keynote Address, OOPSLA '97, Atlanta, GA, October 1997.
20. Harel, David, and Eran Gery, "Executable Object Modeling with Statecharts," *IEEE Computer*, July 1997, pp. 31–42.
21. Hayes, I.J., and C.B. Jones, "Specifications are Not (Necessarily) Executable," *IEE Software Engineering Journal*, November 1989, pp. 330–338.
22. Hasselbring, W., "Animation of Object-Z Specifications with a Set-Oriented Prototyping Language," *Z User Workshop*, June 1994, pp. 337–356.
23. Hazel, Daniel, Paul Strooper, and Owen Traynor, "Possum: An Animator for the SUM Specification Language," Technical Report 97-10, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Brisbane, Queensland, Australia, 1997.
24. Hewitt, M.A., C.M. O'Halloran, and C.T. Sennett, "Experiences with PiZa, an Animator for Z", *ZUM '97: The Z Formal Specification Notation*, Lecture Notes in Computer Science 1221, Springer–Verlag pp. 37–51.
25. Hill, I.D., "Wouldn't It Be Nice If We Could Write Computer Programs in Ordinary English – or Would It?" *The Computer Bulletin*, June 1972, pp. 306–312.
26. Hoare, C,A.R, "An Overview of Some Formal Methods for Program Design," *IEEE Computer*, September 1987, pp. 85–91.
27. Keil, Mark, and Erran Carmel, "Customer-Developer Links in Software Development," *Communications of the ACM*, May 1995, pp. 33–44.
28. Kent, Stuart, "Constraint Diagrams: Visualizing Invariants in Object-Oriented Models," *Twelfth Annual OOPSLA Conference*, Atlanta, GA, October 1997, pp. 327–341.
29. Kilov, Haim, Bernhard Rumpe, and Ian Simmonds, *Seventh OOPSLA Workshop on Behavioral Semantics of OO Business and System Specifications*, TUM-I9820, Technische Universit·t M✧nchen, 1998.
30. Knott, Ron, and Paul Krause, "The Implementation of Z Specifications using Program Transformation Systems: The SuZan Project," *The Unified Computation Laboratory*, C.M.I. Rattray and R.G. Clark (eds.), The Institute of Mathematics and Its Applications, Oxford University Press, 1992, pp. 207–220.
31. Lano, Kevin, and Howard Haughton, *Object-Oriented Specification Case Studies*, Prentice-Hall International, London, UK, 1994.
32. Ledeczi A., Maroti M., Karsai G., Nordstrom G.: "Metaprogrammable Toolkit for Model-Integrated Computing", *Proceedings of the Engineering of Computer Based Systems (ECBS) Conference*, Nashville, TN, March 1999.
33. Lewis, C. S., *Studies in Words*, 2nd ed., Cambridge University Press, Cambridge, England, 1967.
34. Meyer, Bertrand, "On Formalism in Specifications," *IEEE Software*, January 1985, pp. 6–26.
35. Moss, Chris, *Prolog++: The Power of Object-Oriented Programming*, Addison-Wesley, 1994.
36. Naur, Peter, "Programming by Action Clusters," *BIT*, vol. 9, no. 3, 1969, pp. 250–258.
37. Neumann, Peter G., "Only His Only Grammarian Can Only Say What Only He Means," *ACM SIGSOFT Software Engineering Notes*, January 1984, pg. 6.
38. Nordstrum, Greg, Janos Sztipanovits, Gabor Karsai, and Akos Ledeczi, "Metamodeling – Rapid Design and Evolution of Domain-Specific Modeling Environments," *Proceedings of the Engineering of Computer Based Systems (ECBS) Conference*, March 1999.
39. *Object Constraint Language Specification*, Version 1.1, Object Management Group, September 1997.
40. *The Oxford English Dictionary*, 2nd ed., Oxford University Press, Oxford, England, 1989.
41. Petre, Marian, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming," *Communications of the ACM*, June 1995, pp. 33–44.
42. Schach, Stephen R., *Classical and Object-Oriented Software Engineering with UML and C++*, WCB/McGraw-Hill, 1999.
43. Siddiqi, Jawed, Ian Morrey, Graham Buckberry, and Richard Hibberd, "Towards CASE Tools for Prototyping Z Specifications," *CASE '93*, pp. 166–173.
44. Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice-Hall, New York, NY, 1992.
45. Stepney, Susan, Rosalind Barden, and David Cooper, Eds., *Object Orientation in Z*, Springer-Verlag, 1992.
46. Sztipanovits Janos, and Gabor Karsai, "Model-Integrated Computing", *IEEE Computer*, April 1997, pp. 110-112.
47. Utting, Mark, "Animating Z: Interactivity, Transparency and Equivalence," Technical Report No. 94-40, Software Verification Research Centre, Department of Computer Science, The University of Queensland, Brisbane, Queensland, Australia, 1994.

48. Warmer, Jos, and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, Reading, MA, 1999.
49. Warmer, Jos, and Anneke Kleppe, "OCL: The Constraint Language for UML," *The Journal of Object-Oriented Programming*, May 1999, pp. 10–13.
50. West, Margaret M., and Barry M. Eaglestone, "Software Development: Two Approaches to Animation of Z Specifications Using Prolog," *IEE Software Engineering Journal*, July 1992, pp. 264–276.
51. Xiaoping, Jia, "ZANS: A Z Animation System," School of Computer Science, DePaul University, Chicago, Illinois, 1995.