A Graphical Variant Approach to Object-Oriented Modeling of Dynamic Systems

Paul Kinnucan and Pieter J. Mosterman

Abstract— ***Graphical variant modeling*** **refers to a novel approach to object-oriented modeling whereby a class overrides behavior inherited from a parent class by specifying variations in the graphical description of that behavior. This approach differs from conventional approaches wherein object behavior is overridden by replacing the entire description. This paper outlines the potential benefits of graphical variant modeling. It provides a detailed description of an experimental modeler, based on a declarative block diagram language, developed as a test bench for prototyping graphical variant modeling tools and verifying the benefits of the approach in industrial modeling applications. The paper ends with some preliminary conclusions about the potential benefits of graphical variant modeling based on the development and initial usage of the experimental modeler.**

Keywords—**Object-Oriented Modeling; Polymorphism; Inheritance; System Dynamics; Simulation Software Engineering; Embedded Systems; Physical Systems Modeling**

I. Introduction

A bewildering variety of objects inhabit the physical world. The human mind, taking its cue from Aristotle, deals with this complexity by classifying objects based on common properties and behaviors and arranging those classes into hierarchies. Object-oriented modeling systems (OOMS) [1] exploit classification to simplify the development and maintenance of computer models of the physical world.

With such systems, hierarchical sets of classes represent types of components occurring in the modeling domain. The classes define the properties and behaviors of component types. Depending on the OOMS, behavioral descriptions can be procedural as in C++ [2] or declarative as in Modelica™ [3], [4] or a combination of both as in UML™.[1] The behavioral descriptions can take the form of programs or equations as in the case of C++ and Modelica, respectively, or can optionally be graphical (e.g., state charts or sequence diagrams) as in the case of UML.

Subclasses inherit and can extend property and behavioral descriptions from parent classes. In particular, subclasses can define additional properties and behaviors and replace inherited behaviors with substitutes. Inheritance permits reuse of object descriptions, facilitating the efficient creation and maintenance of models.

Typically, the behavior of a subclass is a variant of the parent class and this variation is reflected in the description of the subclass's behavior. This suggests another type of reuse, i.e., reuse of common features of an inherited description in the description that replaces it in the subclass. Procedural modeling languages support a limited form of such reuse. For example, Common Lisp [5] allows a subclass to define before, after, and around methods (i.e., procedural behavior descriptions) to be invoked before, after, or before and after an inherited method. This capability applies only to procedural descriptions and only if the inherited behavior can be incorporated as a unit into the replacement procedure.

This paper explores a more general approach to reuse of inherited behavioral descriptions. With this approach, a subclass that wishes to override an inherited description does so by specifying formal differences between its description and the inherited description. The modeling system applies the differences to the parent description to create the subclass description. In other words, the subclass overrides the parent description by editing rather than replacing it.

The advantage of this approach is generality. In theory, it can be applied to any formal description of a system's behavior, regardless of whether the description is procedural or declarative, textual or graphical. However, the approach raises crucial questions: How useful is it in practice and what tools are needed to support it?

A prototype of a dynamic system modeler, based on The MathWorks™Simulink® [6] and called Object-Oriented Simulink® (OOSL), is being developed to help answer these questions.[2] OOSL implements the description variant approach. In particular, the system enables creation of class hierarchies that define object behavior graphically and in which subclasses can override parent behavior by specifying graphical variants (i.e., differences) between the parent behavior and its behavior.

This paper describes OOSL as it is in its current state and how it can be used to exploit commonalities in graphical descriptions of object behavior. Section II of this paper describes the graphical variant approach to object-oriented modeling and discusses its benefits. Section III provides an overview of Simulink as background for the detailed description of OOSL that follows starting in Section IV. Section IX discusses some insights into graphical variant modeling garnered from developing and using OOSL.

II. Graphical Variant Modeling

Graphic variant modeling refers to an approach to object-oriented modeling whereby a class overrides behavior inherited from a parent class by specifying variations in the graphical description of that behavior.

As motivation for this approach, consider Fig. 1 and

The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760, USA.
E-mail: [paulk|pieter_j_mosterman]@mathworks.com

[1] www.omg.org

[2] OOSL can be obtained by contacting Paul Kinnucan via email at paulk@mathworks.com for more information.

 ISBN # 1-56555-316-0

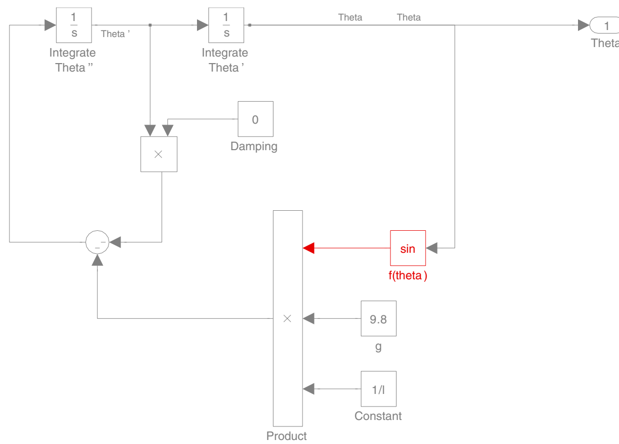

Fig. 1. Nonlinear pendulum diagram.

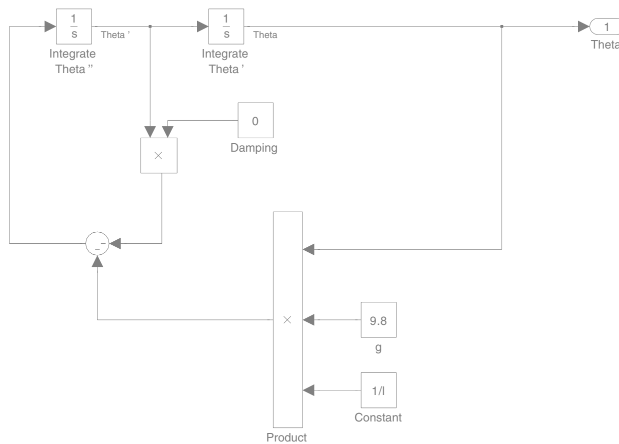

Fig. 2. Linear pendulum.

Fig. 2. The diagram in Fig. 1 graphically represents a second-order nonlinear differential equation that describes the behavior of an idealized ball-and-stick pendulum, i.e., a pendulum consisting of a point mass connected to a frictionless pivot by a massless rod [7]. In short, the diagram describes the behavior of a nonlinear pendulum.

The diagram in Fig. 2 graphically represents a second-order ordinary (i.e., linear) differential equation that closely approximates the behavior of an idealized ball-and-stick pendulum for angles of deflection from the vertical that are less than 15 degrees.

The diagrams are nearly identical, varying only in the presence of the sin block, which computes the sine of its input, in the nonlinear pendulum diagram and the absence of the block in the linear pendulum diagram. This suggests that the nonlinear and linear models of the pendulum might be designed as inheriting and overriding a common diagram from an abstract pendulum class.

Fig. 3 illustrates a possible parent diagram.

It captures the common elements of the linear and nonlinear diagrams. At the single point where they differ, the parent diagram contains a block representing an unspecified (i.e., abstract) function of the pendulum's angle of deflection, theta. The linear and nonlinear pendulum models can now be considered as overriding this abstract function by replacing it with a concrete function as in the case of the nonlinear pendulum model or deleting it altogether as in the case of the linear pendulum model.

Potential advantages of this graphical variant approach to modeling are quicker, more efficent model creation and maintenance. In particular, the approach allows automatic reuse of common graphical elements, thereby eliminating the need to reenter the common elements manually. The approach also speeds maintenance of complex models by allowing graphical changes in parent classes to propagate automatically to subclasses.

The novelty of the approach does raise issues. What tools and changes in modeling practice are required to use it effectively? Are the benefits of the approach sufficient to offset the investment in tools and learning it would require? The following sections describe a prototype graphical variant model tool developed specifically to help answer these questions.

III. Simulink®

Simulink® serves as the basis of Object-Oriented Simulink® (OOSL). This section provides a brief overview of Simulink as an introduction to a discussion of OOSL.

Simulink comprises a model editor and a simulation engine. The model editor is used to enter mathematical models of dynamic systems, for example, sets of ordinary differential equations (ODEs), into the system as block diagrams, which the simulation engine then compiles and executes.

The blocks in a Simulink block diagram represent mathematical functions, implicitly of time and explicitly of other variables called signals. Block connection points represent input and output arguments. Lines connecting the blocks represent input/output relations among the blocks. Blocks are created in a model by copying prototypes from a special

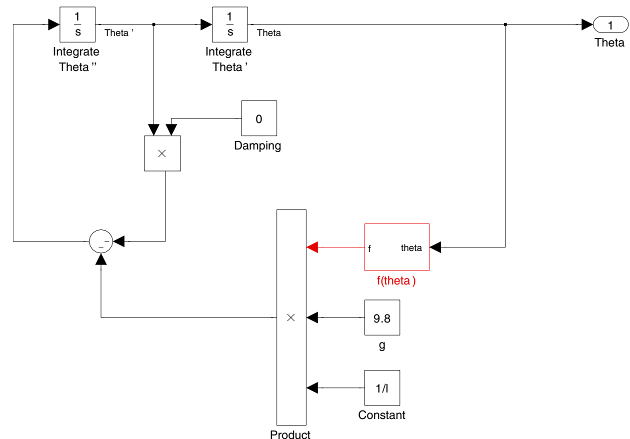

Fig. 3. Abstract pendulum diagram.

type of model called a library.

A block can use one or more parameters (instance variables) to compute its outputs. A user can specialize the behavior of an instance of a block by changing the values of its parameters either programmatically or interactively. Each block has a user interface called a mask. The mask comprises a distinctive symbol, called an icon, that indicates its function and a dialog box that allows a user to set its parameters interactively.

Simulink comes with an extensive set of block libraries that implement commonly used functions. The standard block library includes a set of blocks that enable graphical creation of user-defined blocks. These blocks include the Subsystem, Inport, and Outport blocks.

A Subsystem block is a block whose connection points and behavior are defined by a block diagram stored "inside" the block. Inport and Outport blocks included in the internal block diagram define the Subsystem block's external inputs and outputs. The other blocks in the diagram and the block interconnections define the functional relationship between the Subsystem block's inputs and outputs. The diagram of a standard Subsystem block has one Inport and one Outport. The output of the Inport is connected to the input of the Outport. In other words, the standard Subsystem block outputs its input.

Clicking an instance of a newly created Subsystem block opens its diagram in the model editor. A user can then edit the diagram to create more interesting behavior. Saving the model containing the Subsystem instance saves the edited diagram, making the change in behavior permanent.

A subsystem block's mask (user interface) can be customized, using the Simulink mask editor. The mask editor allows a user to define the block's parameters, dialog box, and icon, using point-and-click operations. Parameter values are stored as values of variables in a subsystem workspace. The parameter variables can be used as the values of parameters of blocks in the subsystem block diagram. This allows the subsystem parameters to determine the behavior of the subsystem.

Simulink is an object-oriented system in the sense that it allows users to create blocks by copying block type exemplars stored in libraries. However, Simulink lacks features more commonly associated with object-oriented systems, such as the ability to define explicitly and formally hierarchies of classes capable of inheriting properties and behavior from their ancestors. For this reason, Simulink might be more accurately called an *object-based* system.

IV. Object-Oriented Simulink®

Object-Oriented Simulink® (OOSL) is a software package that endows Simulink with object-oriented modeling capability. In particular, it allows a user to

- Graphically define a hierarchical set of Simulink block classes, called a *package*, that can inherit their interfaces (inputs and outputs), behavior, parameters, and user interfaces (masks) from their ancestors.
- Create monomorphic and polymorphic instances of classes in a model, i.e., instances that can or cannot change their type during simulation.
- Update instances in models to reflect changes in the classes that define them.

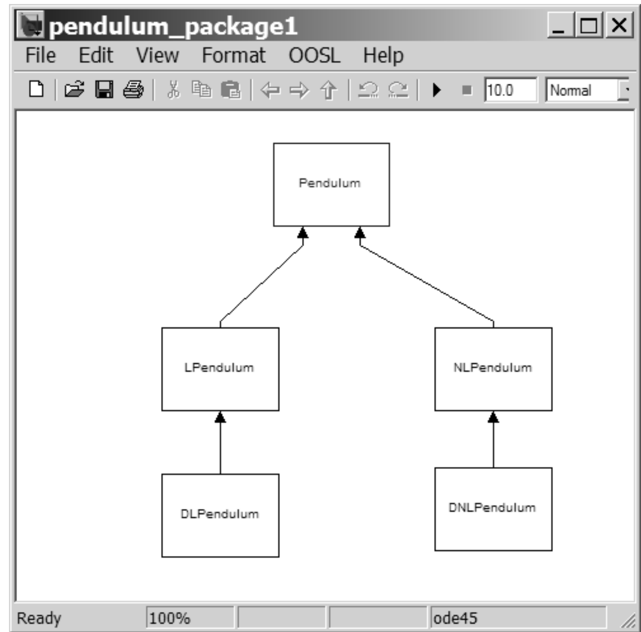

Fig. 4. A package of Simulink®block classes.

To that end, OOSL provides Simulink with a package editor, a specialized version of the Simulink model editor, for creating packages and a set of commands for instantiating them.

OOSL is implemented entirely in M, the MATLAB® [8] programming language, and makes use of the Simulink public M application programming interface (API). It exploits only Simulink features that are available to Simulink users.

OOSL was developed as an experimental tool to be used to explore object-oriented approaches to graphical modeling in Simulink.

V. Packages

A package is a specialized Simulink model that graphically defines a hierarchical set of block classes. A package uses a block diagram to represent the class hierarchy that it defines (see Fig. 4). Each block in the diagram represents a class of Simulink blocks. Lines connecting the blocks represent parent-child relationships among the classes.

OOSL is a single inheritance system. A class can have multiple children but only one parent or no parent. A class that has no parent is called a root class. A descendant of a root class inherits its behavior and properties either directly or indirectly via intermediate classes in the class hierarchy.

A class uses a Simulink block diagram to define the interface (inputs and outputs) and behavior (functional relationship between inputs and outputs). For example, Fig. 5 shows the diagram that defines the interface and behavior

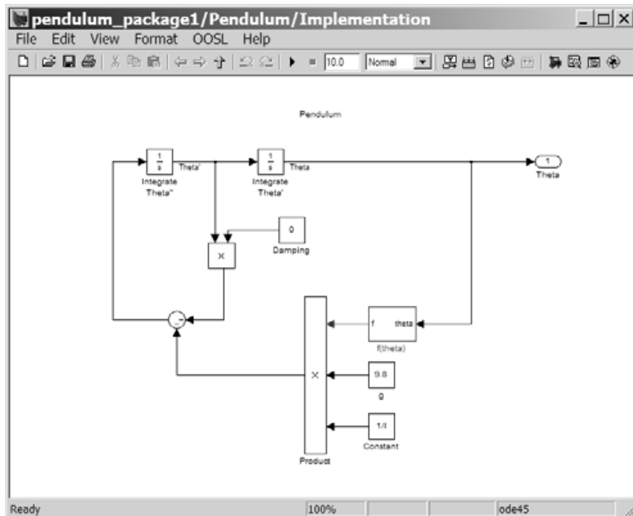

Fig. 5. Instance diagram of the root pendulum class.

(functional relationships between inputs and outputs) of the root pendulum class.

A class uses a mask to define the parameters, parameter dialog box, and icon of the block instances that it defines. Figure 6 shows the parameter dialog box defined by the mask of the root pendulum class.

OOSL implements classes as nested Simulink subsystems. Each class contains a subsystem that defines its instance diagram. The instance subsystem of root class contains the class's instance diagram. The instance subsystem of a descendant class contains the graphical differences between its instance diagram and the diagram it inherits from its parent.

VI. OOSL Workflow

OOSL is intended to support the following workflow for creating and using packages:

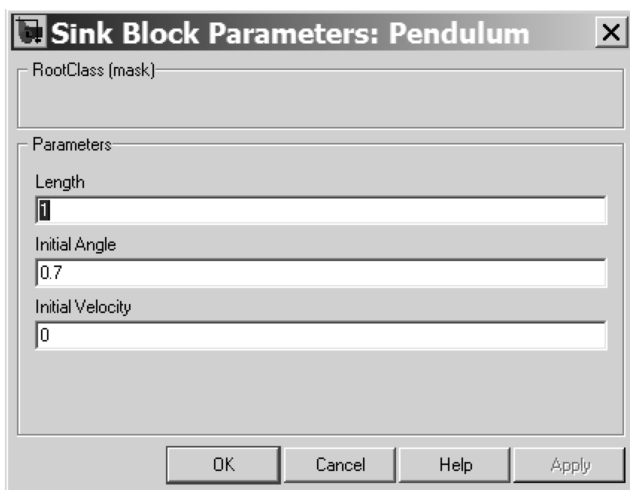

Fig. 6. Root pendulum mask.

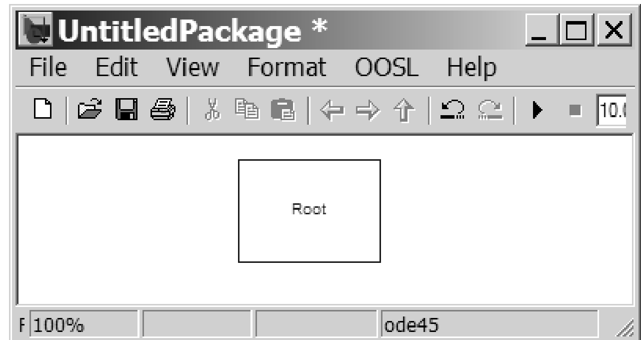

Fig. 7. New package.

1. Create a default package containing a root class having a default instance diagram and instance mask.
2. Edit the root class's default instance diagram and mask to define common block parameters, inputs and outputs, and functional relationships between inputs and outputs.
3. Create a subclass of the root class to define additional parameters, inputs and outputs, and more specialized functions relationships between inputs and outputs. Define the additional inputs and outputs and modified specialized input/outpout relationships by overriding elements of the instance diagram inherited from the root class.
4. Continue elaborating the hierarchy as necessary by creating and editing subclasses of the root or descendant classes.
5. Instantiate classes in a model.

VII. Creating a Package

To create a package, a user starts by selecting the **New Package** command from the **OOSL** menu on the menubar of the OOSL package editor or the Simulink model editor. The **New Package** command creates a package containing a single root class (see Fig. 7).

A. Default Root Class

The default root class includes a default instance diagram comprising a single input connected to a single output (see Fig. 8).

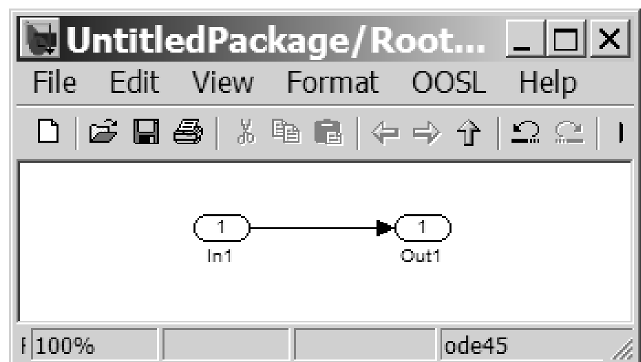

Fig. 8. Default root diagram.

In other words, the default root class defines a block that outputs its input. To define more interesting interfaces and behavior, the user must edit the diagram (see section VII-B). Similarly, to define parameters or customize the appearance of instances, the user must edit the default root's instance mask (see section VII-C).

B. Editing a Root Instance Diagram

To edit a root instance diagram, a user starts by opening the root class in the package editor window. The package editor displays the root instance diagram in a separate editor window. The user can then use standard Simulink editing commands to create, move, copy, and delete blocks in the diagram. A root diagram can include any type of block supported by Simulink, including instances of classes defined with OOSL. To save the changes, the user selects **Save** from the package editor's **File** menu. The package editor saves the edited diagram in the package's *package file*.

C. Editing a Root Instance Mask

To edit a root class's instance mask, a user selects the root class in the package editor and then selects **Edit Mask** from the class's context menu. The package editor displays the class's instance mask in the Simulink mask editor (see Fig. 9).

The mask editor allows the user to define parameters, a parameter dialog box, icon, and documentation for instances of the class. Saving the package saves the edited mask in the package file.

VIII. Creating Subclasses

To create a subclass, the user selects the parent class in the package editor and then selects **Create Subclass** from the **OOSL** menu in the package editor's menu bar. The **Create Subclass** command creates a subclass of the selected class in the package editor window (see Fig. 10).

By default, a new subclass inherits the instance diagram and mask (hence the interface and behavior) of its parent.

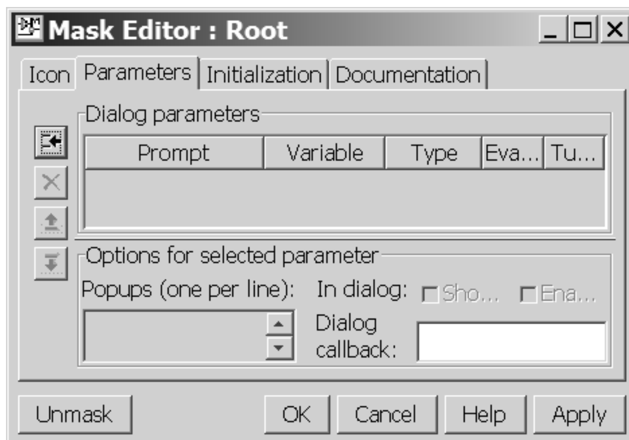

Fig. 9. Mask editor for a default root class.

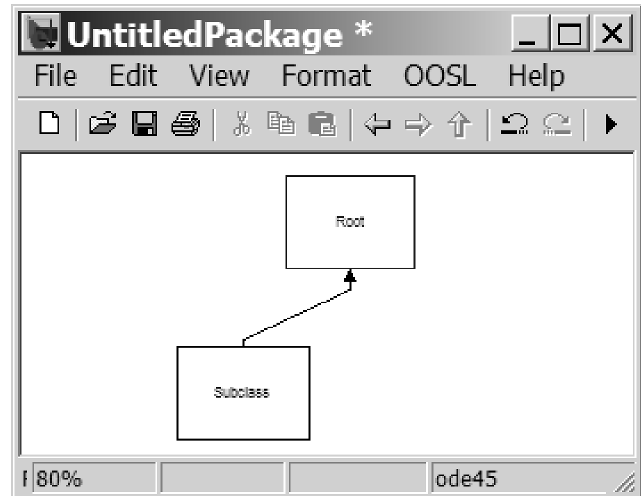

Fig. 10. New subclass.

The user can specialize the subclass by overriding the diagram and mask inherited from the parent class (see Overriding Inherited Diagrams and Extending Inherited Masks, respectively).

A. Overriding Inherited Diagrams

To overide the diagram inherited by a class, the user clicks the class in the package editor. The package editor displays the class's instance diagram, coloring inherited parts of the diagram gray and previously overridden parts red. This enables a user to see at a glance which portions are inherited and which are local. For example, Figure 11 shows the instance diagram of a a newly created subclass as inherited from its parent.

The diagram is grayed out to indicate that it is inherited in its entirety from its parent class. At this point, the user can perform any of the following types of overrides on the inherited diagram:

- **Replace**: replaces a block in the inherited diagram with another.
- **Delete**: deletes a block from the inherited diagram.
- **Insert**: inserts a block between the output port of one block and the input port of another block in the inherited diagram.
- **Add**: adds a diagram to the inherited diagram.

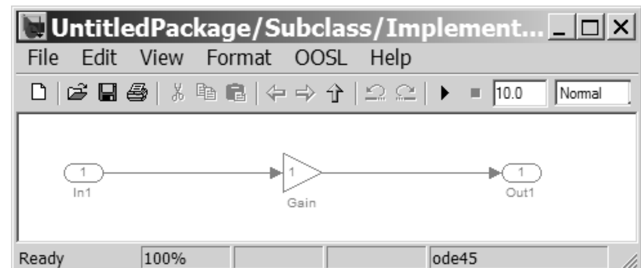

Fig. 11. Inherited diagram.

 ISBN # 1-56555-316-0

A.1 Replacing an Inherited Block

To replace a block in an inherited diagram, the user selects the block and then selects **Replace** from the **Override** menu of the diagram window's **OOSL** menu. The package editor redisplays the inherited diagram in a new window, called the *Replace Command* window, with the selected block highlighted in red (see Fig. 12).

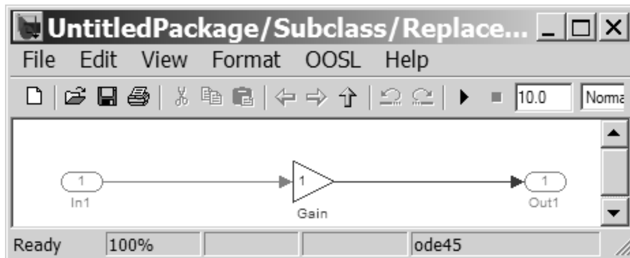

Fig. 12. Replace Command window.

At this point, a user can replace the highlighted block with another block, using standard Simulink block creation and editing commands. For example, a user might replace the Gain block with a Trigonometric Function block, configured to compute the sine of its input, as follows:
1. Delete the Gain block.
2. Create the Trigonometric Function block by dragging it from the Simulink block library browser into the replace command window.
3. Connect the inherited inport and outport blocks to the Sin block.

Closing the Replace Command window saves the change in the package.

Figure 13 shows the instance diagram of the subclass after replacing the inherited Gain block with a Trigonometric Function block configured to perform a sine computation.

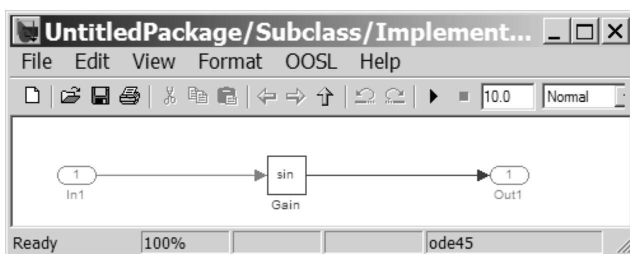

Fig. 13. Result of replacing the inherited Gain block with a Sin block.

The Trigonometric block is colored red to indicate that it is a local override of the diagram inherited from the parent.

A.2 Inserting a Block

To insert a block between the output port of one block and the input port of another block, a user first selects a line connecting the two ports in the inherited diagram. The user then selects **Insert** from the OOSL **Override** menu. The package editor redisplays the inherited diagram in a new window, called the *Insert Command* window, with the line deleted and the two adjacent blocks highlighted in red. The user then creates the block to be inserted in the Insert Command window, connects it to the two ports, and closes the window.

A.3 Deleting a Block

To delete a block, the user selects the block in the inherited diagram and then selects **Delete** from the OOSL **Override** menu. OOSL deletes the block from the subclass's instance diagram.

A.4 Adding a Diagram

To expand an inherited diagram, the user selects **Add** from the OOSL **Override** menu. OOSL displays the inherited diagram (with any previous overrides) in an *Add Command* editor window. The user then creates a new diagram in the editor window, including connections to the inherited diagram.

B. How OOSL Stores Instance Diagrams

The only diagram that OOSL stores in a package is the instance diagram of the root class. For each subclass, OOSL stores a record of the override operations that a user applied to create the subclass's instance diagram. Whenever the instance diagram of a subclass is needed, for example, to display to the user, OOSL reconstructs the instance diagram dynamically by reapplying the recorded overrides to the diagram inherited from the subclass's parents (recursively, if the parent class is not the root class).

Dynamic recreation of the diagrams facilitates propagation of changes in parent classes. A change in the instance diagram of a parent class appears automatically whenever a user displays or instantiates the diagram of a descendant class or updates a model containing instances of descendants of the parent class.

C. Extending Inherited Masks

OOSL allows a user to add parameters to the mask that a class inherits from its parent. To extend an inherited mask in this manner, the user selects the class in the package editor and selects **Edit Mask** from the class's context menu. The package editor displays the inherited mask in the Simulink mask editor.

D. Creating Abstract Classes

A class can be abstract. An abstract class is a class that should not be instantiated because it only partially defines its instances. To specify a class as abstract, a user selects the class in the package editor and then selects **Class Properties** from the **OOSL** menu. OOSL displays a dialog box that allows the user to designate the class as abstract.

E. Specifying Block Output Ranges

OOSL allows a user to specify the range of outputs over which an instance of a class is valid during simulation. A class can specify logical combinations of valid outputs. For

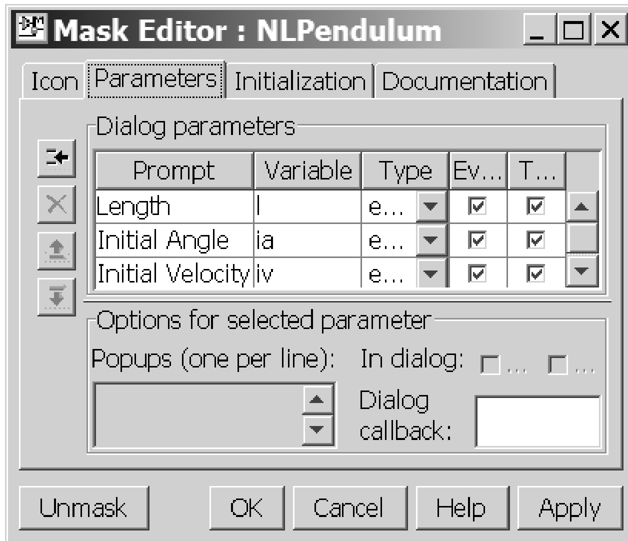

Fig. 14. Mask inherited by NLPendulum class from Pendulum class.

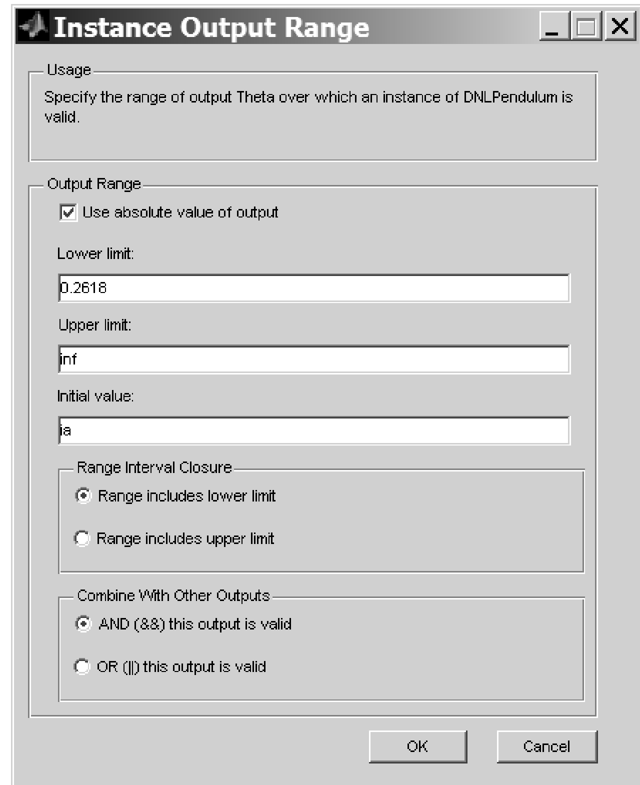

Fig. 15. Output range dialog box.

example, suppose that a class defines a block with two outputs. The class can specify that either of the outputs must be valid or both must be valid for the block to be valid.

This feature is intended primarily to support polymorphic instantiation of classes. However, in future OOSL versions, it might also support automatic range checking during simulation of monomorphic instances.

To specify a valid range for a given output, a user selects the corresponding outport block in the class's instance diagram and then selects **Output Range** from the **Enable Instance** submenu of the package editor's **OOSL** menu. An output range property dialog box appears (see Fig. 15).

The dialog box allows the user to specify the appropriate range properties for the output, including how its validity should be combined logically with those of other outputs to determine the overall validity of the block.

F. *Instantiating Classes*

A user instantiates a class in a model by dragging it from the package editor window into a Simulink model editor window that displays the model. Instances created by OOSL are fully compatible with Simulink and support all Simulink features.

OOSL supports several types of instantiation, depending on whether the class is concrete or abstract and whether it specifies the output range of its instances.

F.1 Monomorphic Instantiation

If the class is concrete, OOSL creates a monomorphic instance of the class in the model, i.e., an instance that cannot change type during simulation. OOSL creates the instance in the model as a Subsystem block having the mask and instance diagram specified by the class. OOSL constructs the instance diagram and the mask by recursively applying the diagram and mask variants specified by the class and its ancestors to the root diagram and mask.

F.2 Polymorphic Instantiation

If the class is abstract and contains two or more concrete descendants having the same interface (i.e., the same inputs and outputs), OOSL creates a polymorphic instantiation of the class in the model, i.e., an instance that can change type during simulation. If the concrete descendants specify valid output ranges, OOSL creates a polymorphic instance that changes type depending on its output.

Figure 16, for example, shows a model containing a polymorphic pendulum block created by dragging the Pendulum class shown in Fig. 4 into the model.

The model also contains monomorphic instances of the damped linear pendulum (DLPendulum) and damped nonlinear pendulum (DNLPendulum) classes instantiated by dragging their respective classes from the package editor into the model. The Pendulum block operates as a damped linear pendulum (DLPendulum) for deflection angles less than 15 degrees and as a damped nonlinear pendulum for angles greater than or equal to 15 degrees. The other blocks instantiate their respective concrete types over the entire pendulum range.

Figure 17 shows the outputs of the three pendulums. The scope displays only two traces. This is because the output of the polymorphic (Pendulum) and damped nonlinear (DNLPendulum) block coincide. In other words, the polymorphic pendulum combines the computational efficiency of the linear pendulum with the accuracy of the nonlinear

 ISBN # 1-56555-316-0

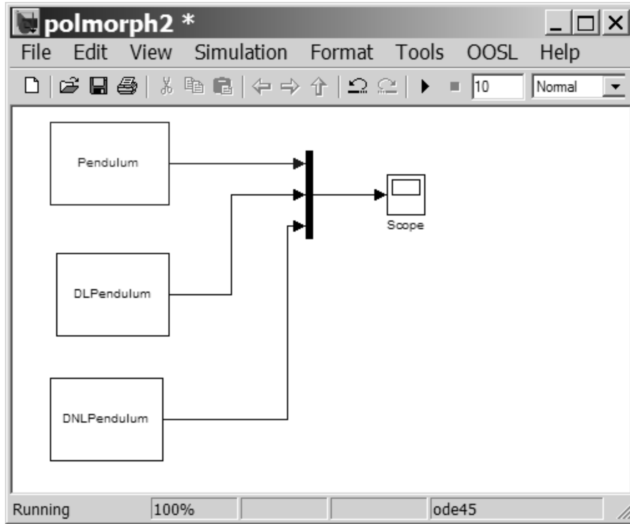

Fig. 16. Model containing a polymorphic pendulum block.

pendulum.

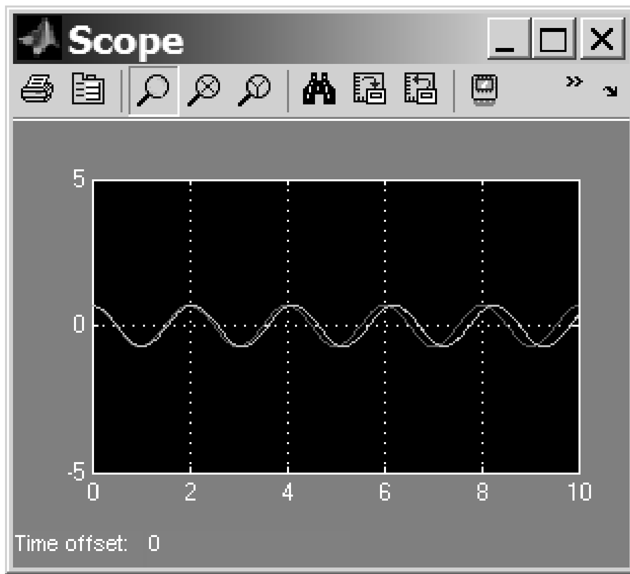

Fig. 17. Outputs of polymorphic and monomorphic pendulum blocks.

If an abstract class does not specify mutually exclusive ranges for its outputs, OOSL instantiates the class as a polymorphic instance with an external type input. This allows the model to determine the actual type of the polymorphic block at run time.

Figure 18, for example, shows a model containing a polymorphic pendulum block with an external input. As in the previous example, the model also contains monomorphic instances of the damped linear and damped nonlinear classes. The external Type Selector block selects the block's type as either 1 (=DLPendulum) or 2 (=DNLPendulum), depending on the pendulum's output angle.

The model displays the type selector on a scope along with the outputs of the pendulums (see Fig. 19).

G. *Updating Models to Reflect Class Changes*

If a user changes a class, models that use instances of this class must be updated to reflect the changes in the class. OOSL provides a command, accessible from the **OOSL** menu in the Simulink model editor, to do this. This command currently reinstantiates all instances in the model. Future versions will instantiate only instances whose classes have changed since the instances were created and do so automatically when a user initiates simulation of the model.

IX. OOSL Experience

As a test of its usability, OOSL has been used to reimplemented a set of Unit Delay blocks from the Simulink Additional Discrete blocks library as a package of Unit Delay classes (see Fig. 20).

An advantage of the class implementation is that it reveals graphically the hierarchical relationships among the various types of unit delay blocks. It also facilitates creation of additional types derived from existing types.

X. Conclusions

Graphical variant modeling is a new and promising approach to object-oriented modeling. By facilitating reuse of common elements of graphical descriptions of object behavior, it significantly decreases the time required to create new models and maintain existing models. A prototype graphical variant modeling tool, OOSL, has been developed and completely integrated with Simulink. Future research

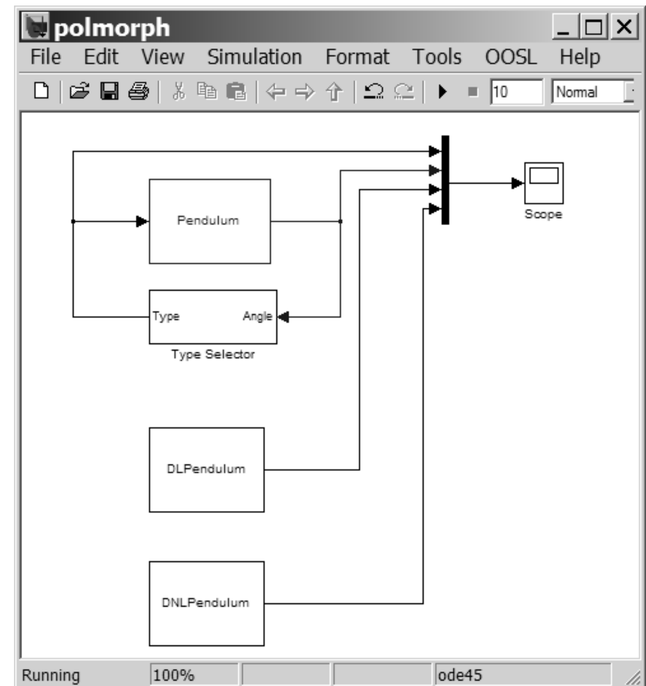

Fig. 18. Model containing a polymorphic pendulum block with an external type input.

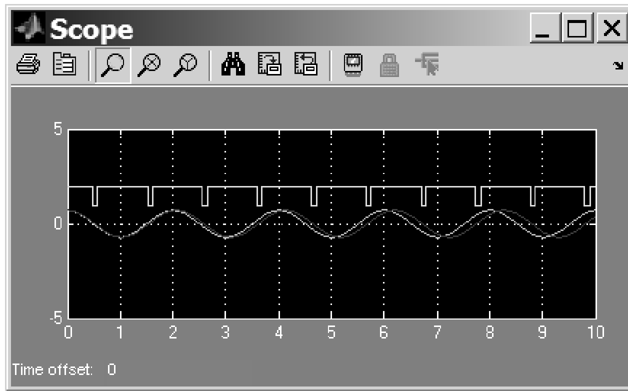

Fig. 19. Outputs of polymorphic and monomorphic pendulum blocks.

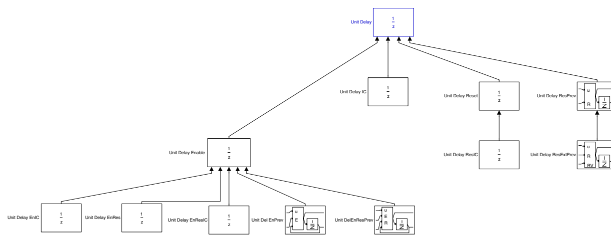

Fig. 20. Unit delay package.

intends to use OOSL to explore approaches to tool design and determine the feasibility of the approach in real-world modeling applications. Experience with OOSL could help to determine over the next few years whether the benifits of this approach are actually sufficient to justify its use in industrial modeling applications.

Acknowledgments

MATLAB and Simulink are registered trademarks and The MathWorks is a trademark of The MathWorks, Inc. Modelica is a trademark of the Modelica Association. UML is a trademark of the Object Management Group, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.

Copyright © The MathWorks, Inc., 2007.

References

[1] A. Goldberg, *Smalltalk-80: The interactive programming environment*. Reading, Massachusetts: Addison-Wesley, 1984.
[2] B. Stroustrup, *The C++ Programming Language*. Reading, Massachusetts: Addison-Wesley Publishing Company, 2000, iSBN 0-201-70073-5.
[3] M. M. Tiller, *Introduction to Physical Modeling with Modelica*. Boston: Kluwer Academic Publishers, 2001, iSBN 0-7923-7367-7.
[4] H. E. *et al*, "Modelica$^{\text{tm}}$–a unified object-oriented langauge for physical systems modeling: Language specification," http://www.modelica.org/, Dec. 1999, version 1.3, http://www.modelica.org/.
[5] G. L. Steele, *Common Lisp The Language*, 2nd ed. Woburn, Massachusetts: Digital Press, 1990, iSBN 0-201-70073-5.
[6] Simulink, *Using Simulink*, The MathWorks, Inc., Natick, MA, Mar. 2006.
[7] R. E. Williamson, *Introduction to Differential Equations and Dynamical Systems*, 2nd ed. New York, New York: McGraw-Hill, 2001, iSBN 0-07-232573-9.
[8] MATLAB, *The Language of Technical Computing*, The MathWorks, Inc., 2004.