SIMBIOS: SIMulation Based on Icons and ObjectS

Antoni Guasch
Departament de Enginyeria de Sistemes, Automàtica i Informàtica Industrial & Institut de Cibernètica
Universitat Politècnica de Catalunya
08028 Barcelona (SPAIN)

Visiting Research Professor
Department of Computer Science
California State University, Chico
Chico, California 95929-0410

Paul A. Luker
Department of Computer Science
California State University, Chico
Chico, California 95929-0410

ABSTRACT

An interactive simulation environment prototype for the study of continuous models has been implemented using the Smalltalk/V system. Models are designed in a graphical manner and can be directly executed without preprocessing, compiling and linking. Therefore, models can instantaneously react to new configurations. Owing to the segmentation of the submodel code, the system supports hierarchical simulation design and execution without restrictions. A directed graph, which reflects the computational flow, is used to direct the interactive experiments.

1 INTRODUCTION

When working with simulation languages, however "friendly", a user has to write a program. For many, programming is a process which is too far removed from the way that they normally express and think about their particular application. What is more, the development of a large simulation in small, well-defined stages is usually not easy in a programming environment. It is much more satisfactory to have an interface with the simulation software which is at the application level, and which also supports the easy production of large simulations from smaller ones.

During the last decade a number of developments has accelerated the evolution of the simulation environment, which is intended to provide integrated software support for many aspects of the modelling and simulation process. Among these developments are the use of multiple windows, menus and icons, and the more widespread recognition of the value of object-oriented programming. The use of artificial intelligence techniques, particularly expert systems, has also added much power and convenience by facilitating the so-called intelligent simulation environment [Luker 1989]. This particular development is not our concern here, although it will be addressed in our subsequent work.

In this paper, we describe a prototype of a highly flexible and user-oriented simulation environment, SIMBIOS. Although SIMBIOS takes full advantage of icons, menus and windows for presenting a clear interface to the user, it should be stressed that object-orientation is the single most important characteristic of the software. Without this, it would be extremely difficult to produce a system which is so flexible and adaptable, and it would have taken much longer to develop. SIMBIOS is implemented in SMALLTALK/V.

1.1 Outline of SIMBIOS

From the outset, it was taken for granted that SIMBIOS should support the hierarchical development of simulations. A simulation system should facilitate the specification, execution and testing of submodels, and allow larger models to be composed from submodels. In turn, these larger models may themselves be designated submodels and used in even larger simulations. The theoretical groundwork for piecewise continuous submodel internal representation and sorting is described in [Guasch 1987 and 1990]. At present, only continuous models are handled, however, further extensions to include discrete and combined simulation are planned.

The interface, as has been mentioned, is icon-based. Each element of a simulation, whether a primitive element such as an integrator, or a submodel such as a distillation column, will have its own icon. A simulation is constructed by specifying icons and connecting them as required.

1.2 The rôle of objects

Object-oriented programming (OOP) became generally available with the release of SIMULA in 1967 [Birtwistle 1973]. However, it took just under twenty years for the virtues of the paradigm to be appreciated. Today, OOP and, more importantly perhaps, object-oriented design (OOD), are seen as essential tools for overcoming the software crisis. There is now a considerable literature which extols the virtues of the object-oriented approach, with [Meyer 1988] being, as yet, the best single source of ammunition. Our intention here is simply to summarize the main advantages to software engineering of OOD and OOP, and then to relate this specifically to simulation and project SIMBIOS.

For very many software applications, OOD presents a more natural way of describing the problem, with a one-to-one relationship between real-world objects and modules. Subsequent changes are much more readily integrated into an object-oriented design than a top-down one. If the OO design is translated into an OO program, these benefits are inherited by the software itself. The flexibility that obtains is quite remarkable.

With software maintenance accounting for a very high proportion (70% according to Meyer) of software cost, this flexibility is very important. It manifests itself in two ways in particular: **extensibility**, the capacity to be extended and altered throughout the software life cycle, and **reusability**, a measure of the extent to which code from one application can be utilized in another. Objects are very amenable to reuse.

Some of the characteristics of object-oriented software which contribute to maintainability are the high level of **abstraction** that can be achieved (more so in some languages than others) coupled with **inheritance** through the class/sub- class hierarchy. Abstraction enables the general properties of a large set of objects to be represented once, and only once, at an ancestral level in the class hierarchy. These general properties are inherited by the descendants, who can add their own characteristics and even over-ride inheritance when this is appropriate. Abstraction therefore encourages a focus on properties at an appropriate point of the design and coding. The reusability of abstract classes is high.

A companion to abstraction is **polymorphism**, which enables different objects to be treated in the same way. If we want to print the name of an object, MyObject then it is nice to be able to say something like: MyObject printName, in the knowledge that this will work, regardless of what kind of object MyObject represents.

It is not insignificant that it was a need to be able to describe (discrete event) simulations in a natural way that led to the design of SIMULA. However, the language designers had the wisdom and foresight to make SIMULA a general purpose programming language, complete with object-oriented features. Any simulation is a software representation of real world objects. What could be more natural than to mirror the real world with software objects?

As with any software, simulations have to be maintained, and extensibility (as well as reusability) is highly desirable. Class hierarchies provide a natural framework for accommodating submodel hierarchies. Abstraction and inheritance should be used to the maximum extent. In a continuous system simulation, for example, there will be certain properties that **all** dynamic objects will share, whatever the nature of the particular submodel in question.

The relationship between objects and WIMP (Window/Icon/Menus that Pop up) interfaces is very strong. Apple has long used an object-oriented approach on the Macintosh, and now NeXT and Hewlett-Packard have object-oriented interfaces. Again, it is abstraction and inheritance that come to the rescue. For example, all windows have the same general characteristics, so it makes sense to only describe a window once, as a class. Each time we want a new window to be drawn, it is simply a matter of creating a new instance (object) of a window with the desired parameters.

In summary, without OOP, project SIMBIOS would be an entirely theoretical undertaking!

1.3 The implementation language

One of the authors has produced a series of implementations of his ideas. The first of these was in C, and then C++ was used. The move to C++ provided a much more productive environment than the original one, thanks entirely to the object-orientation. C++ does not come equipped with a library of pre-defined classes, therefore the programmer has to be concerned with details at a lower level than is necessary in Smalltalk (and Objective-C.) As object-oriented languages go, C++ is really little better than C. However, C++ lacks dynamic (i.e. run-time) binding, which maximizes the level of polymorphism (and thereby abstraction) that can be obtained.

C++ was inspired by SIMULA; so was Smalltalk. However, Alan Kay's aim was to improve upon the inspiration, not just transplant it into a different base language. Smalltalk has evolved into an extremely powerful software development environment. It comes with well over one hundred pre-defined classes, which form the basis for all the interface operations, i/o and data structuring you are ever likely to need. What doesn't suit, you modify, or add your own subclass. Smalltalk does support dynamic binding, at some expense in terms of run-time efficiency, it must be added.

SIMBIOS is currently implemented in Smalltalk/V on IBM and Macintosh platforms.

1.4 Overview of paper

The rest of this paper is concerned with some of the details of SIMBIOS, and begins with an overview of the submodel/model structure. Subsequent sections address the experimentation mechanisms and the class hierarchy. The paper concludes with a discussion and the conclusions the authors have drawn from their work to date.

2 SUBMODEL STRUCTURE

A SIMBIOS simulation submodel (class) holds the dynamic code needed to perform a simulation experiment and static information needed for the management of the submodel during the different phases of a simulation study. For example, a submodel class stores three levels of representation that can be used by the simulation environment to perform modelling-related tasks:

1) The **submodel graphical representation** includes the submodel icon representation (class *SubmodelIcon*) plus the representation of the input and output variables (*SymbolIcons*).

2) The **submodel icon digraph** (class *SubmodelIconDigraph*) is a digraph that relates the submodel output symbol vertices (class *OutputSymbolVertex*) to the submodel input symbol vertices (*InputSymbolVertex*) through the submodel icon vertex (class *SubmodelVertex*). The submodel icon digraph is used to specify the model icon digraph (*IconDigraph*) introduced later.

3) The **segment-link digraph** (class *SegmentLinkDigraph*) shows the sorting relationship between submodel segments and is used to specify the model digraph (*ModelDigraph*) introduced later. The submodel dynamic code is grouped into several segments (submodel segments) to allow a clean structuring of the hierarchical model. Furthermore, in object-oriented programming, each segment is implemented as an instance method. There are six possible submodel segments: initial, discontinuous, state, algebraic, derivative and output segments [Guasch 1990]. The initial segment has to be called to initialize the submodel before a simulation run; the discontinuous segment computes the discontinuity functions; the output segment has to be executed at each communication point; and the state, algebraic and derivative segments are needed to compute the submodel derivatives.

3 MODEL STRUCTURE

This first SIMBIOS prototype only supports the modelling, simulation and experimentation of piecewise continuous models. A model is composed of a hierarchical set of submodels. However, a model can itself be a submodel within some larger model. Therefore, a model is a relative concept, which depends on the experiment being performed. In this study, the model is the particular submodel that we are trying to specify by using pre-defined submodels and experiments *via* the graphical modelling interface.

The graphical interface and, in particular, the icon modeling pane (*SimbiosIconPane*) will support several modeling methodologies, such as those based on bond-graphs, analog computation, control system theory and general system theory. Therefore, the user will be able to choose the modeling formalism that best suits the application. Although the graphical interface is an important aspect of the system, two more model representations are needed in order to manipulate the graphical representation of the model and to guide the dynamic execution of that model (see Figure 1):

- An **icon digraph** (class *IconDigraph*) that holds topological information about the graphical representation.

- A **model digraph** (class *ModelDigraph*) that represents the sorting relationship between called submodel segments and therefore reflects the computational flow of the model.

These three model representation levels are not independent. They interact continuously throughout all the graphical model editing operations. In general, every intended graphical operation is broadcast first to the *IconDigraph* level and then, to the *ModelDigraph* level. Each level checks the validity of the operation and performs internal changes related to that operation. If the editing operation is incorrect it is cancelled. For example, if we want to connect two submodel icons through input/output symbol icons, the *IconDigraph* will check that any connecting path icon segment does not overlap with the other path segments, and the *ModelDigraph* will check that this connection does not provoke an implicit loop in the model code. Moreover, if the connection is authorized, the *IconDigraph* and the *ModelDigraph* will update their internal model representations to reflect the new connection.

3.1 Graphical level (*SimbiosIconPane*)

This level administers three graphical elements: **submodel icons** (*SubmodelIcon*), **input and output symbol icons** (*SymbolIcon*), and **connections** between input and output symbol icons (*PathIcon*). All these elements are active, therefore, each one will answer according to its class type when it is selected with a mouse. This active behavior is supported by the *IconDigraph*. Besides broadcasting any desired graphical operation to the *IconDigraph* and to the *ModelDigraph*, the operation is also broadcast to an editor controller (*IconEditor*) which may authorize or deny the intended menu operation. This feature can be used, for example, to guide the user through a set of training steps.

The following basic *SubmodelIcon* menu-editing operations have been implemented:

- **insert** the submodel icon in a chosen position.
- **move** the submodel icon to a new position. If it is already connected to any other submodel icons, the corresponding connecting *PathIcons* have to be updated automatically.
- **delete** the submodel icon. Again, if it is connected, the attached *PathIcons* have to be deleted automatically.
- **connect** one of its associated input or output *SymbolIcons* to an output or input *SymbolIcon* through a *PathIcon*.
- **disconnect** one of its associated input or output *SymbolIcons*.

So far, little has been said about the input/output *SymbolIcons*. A *SymbolIcon* has a default position with respect to the virtual origin of its *SubmodelIcon*. Furthermore, a *SubmodelIcon* drags its associated *SymbolIcons* through its editing operations. Therefore, when a *SubmodelIcon* is moved to a new position, the associated *SymbolIcons* are also moved.

A *SymbolIcon* can be moved, connected or disconnected, but it cannot be inserted or deleted directly. These operations are performed indirectly when the associated *SubmodelIcon* is inserted or deleted. Moreover, a *SymbolIcon* can only be moved to a position adjacent to or inside the associated *SubmodelIcon*. Finally, a *PathIcon* can ònly be deleted.

New menu-editing operations will be added in the future. Additionally, the graphical icons support information retrieval operations not mentioned here. At present, the *FreeDrawing* class included in the Smalltalk system is used to specify the *SubmodelIcon* graphical representations. However, a more specialized class is preferable.

3.2 Icon digraph (*IconDigraph*) level

A **model icon digraph** is a directed graph I=(V,E), where:

V = (Vi,Vs,Vp) is the set of vertices, such that:

- Vi is a set of **submodel vertices** (class *SubmodelVertex*). Each *SubmodelIcon* has an associated *SubmodelVertex*.
- Vs is a set of **symbol vertices** (class *SymbolVertex*) associated with the *SymbolIcons* (input/output variables). Two types of symbol vertices can be defined; output symbol vertices (Vs_o), represented by class *OutputSymbolVertex*, associated with output symbol icons (*OutputSymbolIcon*) and input symbol vertices (Vs_i) (*InputSymbolVertex*) associated with input symbol icons.
- Vsp is a set of **path vertices** (class *PathVertex*). Each graphical connection between an output symbol vertex and an input symbol vertex has an associated path vertex.

The following properties are intrinsic to model icon digraphs:

- The predecessors of a submodel vertex are instances of *InputSymbolVertex* and the successors are instances of *OutputSymbolVertex*. Therefore, for all $v_i \in Vi$:

  $\Gamma^{-1}(v_i) \in Vs_i$ and $\Gamma(v_i) \in Vs_o$.

- The predecessor of an output symbol vertex is a single instance of *SubmodelVertex* and the successors are instances of *PathVertex*. Therefore, for all $v_i \in Vs_o$:

  $\Gamma^{-1}(v_i) \in Vi,\ \Gamma(v_i) \in Vp,$

  $|\Gamma^{-1}(v_i)| =$ 1 (if it is connected)
  0 (if it is not connected)

- A *PathVertex* has a single *InputSymbolVertex* as a predecessor and a single *OutputSymbolVertex* as a successor. Therefore, for all $v_i \in Vp$:

  $\Gamma^{-1}(v_i) \in Vs_i, \Gamma(v_i) \in Vs_o, |\Gamma^{-1}(v_i)| = 1,\ |\Gamma(v_i)| = 1.$

- An *InputSymbolVertex* has a single *PathVertex* as a predecessor and a single *SubmodelVertex* as a successor. Therefore, for all $v_i \in Vs_i$:

  $\Gamma^{-1}(v_i) \in Vp, \Gamma(v_i) \in Vi, \Gamma(v_i)| = 1,$

  $|\Gamma^{-1}(v_i)| =$ 1 (if it is connected)
  0 (if it is not connected)

The *IconDigraph* class supports the graphical editing operations described in the previous subsection. Therefore,

- When a submodel icon (and its associated input/output symbol icons) is inserted in the *SimbiosIconPane*, the associated submodel icon digraph is inserted into the icon digraph. Moreover, the icon digraph checks that the submodel icon does not overlap any other submodel icon, symbol icon or path icon.

- In a moving operation, the icon digraph checks that the submodel icon does not overlap any other submodel icon, symbol icon or path icon.

- When an output symbol icon is connected to an input symbol icon through a path icon, the associated path vertex and the related directed edges are inserted in the icon digraph. Moreover, the icon digraph is used to check that the input symbol icon is not yet connected and that the proposed path icon does not overlap with other path icons, symbol icons or submodel icons. When a submodel icon is deleted, the icon digraph is used to check whether or not it is connected. If it is, the connecting path icons are also deleted. In addition, the associated edges and vertices are removed from the icon digraph.

Other graphical editing operations, have a complementary effect in the icon digraph. In conclusion, the main purpose of the icon digraph is to support the *SimbiosIconPane* model and experiment editing operations.

3.3 Model digraph (*ModelDigraph*) level

The model digraph concept explained here and used in SIMBIOS is a simplification of the submodel digraph concept introduced in [Guasch 1990]. A model digraph G=(V,E) is a directed graph, where:

V = (Vs,Ve) is the set of vertices, such that:

- Vs is the set of **symbol vertices** as defined in the previous section.

- Ve is a set of **executable vertices** (class *ExecutableVertex*). Each executable vertex is associated with a call to a submodel segment.

Different subsets of Ve can be defined:

- Ve_d: **derivative vertices** (class *DerivativeVertex*); each call to a lower level derivative segment has an associated derivative vertex.

- Ve_{gs}: **discontinuous vertices** (class *DiscontinuousVertex*) associated with calls to lower level discontinuous segments.

- Ve_s: **state vertices** (class *StateVertex*), each call to a lower level state segment has an associated state vertex.

- Ve_n: **initial vertices** (class *InitialVertex*), each call to a lower level initial segment has an associated initial vertex.

- Ve_t: **output vertices** (class *OutputVertex*), each call to a lower level output segment has an associated output vertex.

Although each graphical editing operation invokes an equivalent action in the *ModelDigraph*, there is not a straightforward relationship between it and the graphical model. This is because the *ModelDigraph* is concerned with the computational flow and not with the graphical model. Therefore, a vertex equivalent to the *PathVertex* does not exist. An *OutputSymbolVertex* is directly connected to an *InputSymbolVertex*. Moreover, instead of using a single vertex associated with each *SubmodelIcon*, the number of vertices depend on the number of submodel methods (segments). The latter information is retrieved from the *SegmentLinkDigraph* explained earlier.

The *ModelDigraph* has two main rôles. First, it is employed in the translation of the model into a submodel class which can later be reused as a submodel in a bigger model. Second, it can be used directly to perform an interactive experiment.

4 EXPERIMENT MANAGER (*SymbolicExperiment*)

Before the execution of a *SymbolicExperiment*, the executable vertices have to be grouped in sorted order into four different segments (arrays):

- *initialSegmentArray* contains the executable vertices needed to initialize the experiment (model). The associated code is only executed once before every simulation experiment.

- *odeSegmentArray* (ordinary differential equation segment) contains the executable vertices needed to computed the experiment (model) derivatives.

- *discontinuousSegmentArray* contains the executable vertices needed to compute the experiment (model) discontinuity functions.

- *outputSegmentArray* clusters the executable vertices whose associated code needs to be executed only at communication points.

When a *SubmodelIcon* is inserted in the *SimbiosIconPane*, instances of the associated submodel and submodel input/output variables are also created and stored in the simulation environment. Furthermore, each executable vertex of the *ModelDigraph* holds a pointer to its associated submodel instance and stores the symbolic name needed to call the associated submodel segment. Moreover, each input/output *SymbolVertex* holds a pointer to its associated input/ouput variable instance.

Therefore, to execute a simulation experiment (class *SymbolicExperiment*) interactively, the segment code associated with each executable vertex has to be executed in a sorted order. After the 'execution' of a submodel segment (*ExecutableVertex*), its output values are propagated through the *ModelDigraph* to the input of other submodel segments.

Every time a change is made to the graphical model (experiment), the *ModelDigraph* is updated in an equivalent manner. Therefore, the only task left to be performed before a simulation experiment commences, is to split the executable vertices and group them into the previous segment's arrays. This is a rapidly executed task which is not significant in terms of computation time.

5 CLASS HIERARCHY

Although Table 1 includes all the classes mentioned in this paper, it only represents a partial list of the classes designed for the SIMBIOS simulation environment. The *Pane*, *SubPane*, *GraphPane* and *IconPane* classes belong to the Smalltalk/V (286 and MAC) environments.

6 CONCLUSIONS

The design and implementation of project SIMBIOS to date have clearly highlighted the value of object-oriented design and programming. The system can be amended and be operational again in a time which has staggered the onlooker. It is particularly pertinent to have a software environment which is so supportive of an experimentally-oriented approach to programming, which, of course, simulation so often is.

Smalltalk has been found to be a far superior and higher level development **environment** than C++, which remains no more than just a language. Being interpreted, execution is not always as rapid as one might like, however, it is nowhere near as unacceptable as some would have you believe. For those who must have speed, compilation, automatic translation (to Objective-C) or specialised hardware are options.

SIMBIOS itself is a very powerful, yet flexible, platform on which we can build incrementally. It is our intention to enhance the capabilities of SIMBIOS by adding further experimentation features, by moving towards the incorporation of discrete event elements, and by adding "expertise" in the form of expert system rules.

7 ACKNOWLEDGEMENTS

The authors are grateful to the support provided by research grants from: the Ministerio de Educación y Ciencia of Spain; the Fulbright Institution; the Graduate School of California State University, Chico; Apple Computer Inc.

- *IconEditor*
- *Pane*
  - *SubPane*
    - *GraphPane*
      - *IconPane*
        - *SimbiosIconPane*
- *Icon*
  - *SubmodelIcon*
  - *SymbolIcon*
  - *PathIcon*
- *Graph*
  - *DirectedGraph*
    - *IconDigraph*
      - *SubmodelIconDigraph*
    - *ModelDigraph*
      - *SegmentLinkDigraph*
- *Vertex*
  - *SimbiosVertex*
    - *ExecutableVertex*
      - *InitialVertex*
      - *DiscontinuousVertex*
      - *StateVertex*
      - *AlgebraicVertex*
      - *DerivativeVertex*
      - *OutputVertex*
    - *PaneVertex*
      - *SubmodelVertex*
      - *PathVertex*
      - *SymbolVertex*
        - *InputSymbolVertex*
        - *OutputSymbolVertex*
- *EnvironmentElement*
  - *Submodel*
    - :::::::::
    - *Experiment*
      - *SymbolicExperiment*
  - *Variable*

Table 1. Partial SIMBIOS class hierarchy.

8 REFERENCES

Birtwistle, G.M.; O-J. Dahl; B. Myrhaug and K. Nygaard. 1973. *SIMULA begin*. Studentlitteratur.

Guasch, A. 1987. "MUSS: A contribution to the structural analysis of continuous system simulation languages", Ph.D Thesis, Universitat Politècnica de Catalunya, Barcelona, Spain.

Guasch, A. and R.C. Huntsinger . 1989. "Object-oriented continuous system simulation". *Proceedings of the Summer Computer Simulation Conference* (Austin, Texas,July 1989). SCS. pp 562-565.

Guasch, A. 1990. "Submodels in Hierarchical Continuous Simulation". *Transactions of the Society for Computer Simulation.* To be published.

Luker, P.A. 1989. "Intelligent Simulation Environments." In *IMACS Annals on Computing and Applied Mathematics,* Volume 2. Eds. R. Huber, X. Kulikowski, J.M. David and J.P. Krivine. J.C. Baltzer AG, Basel.

Meyer, B. 1988. *Object-oriented Software Construction.* Prentice-Hall. Series Ed. C.A.R. Hoare.

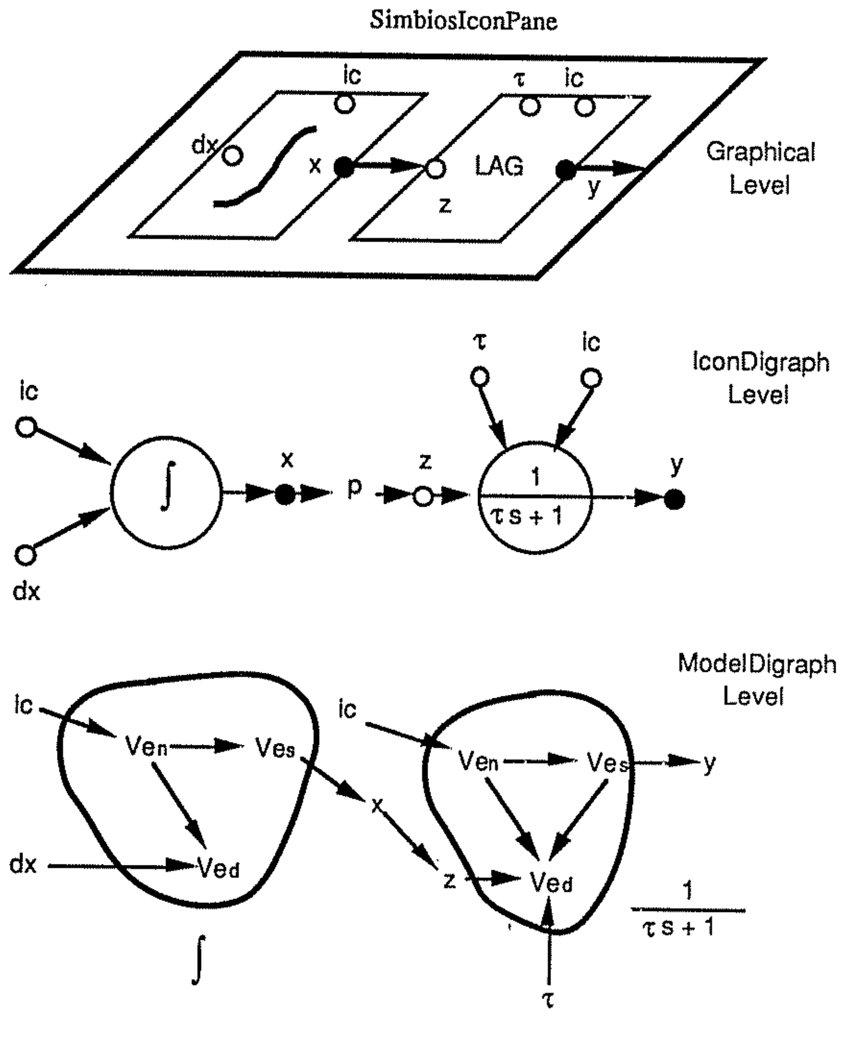
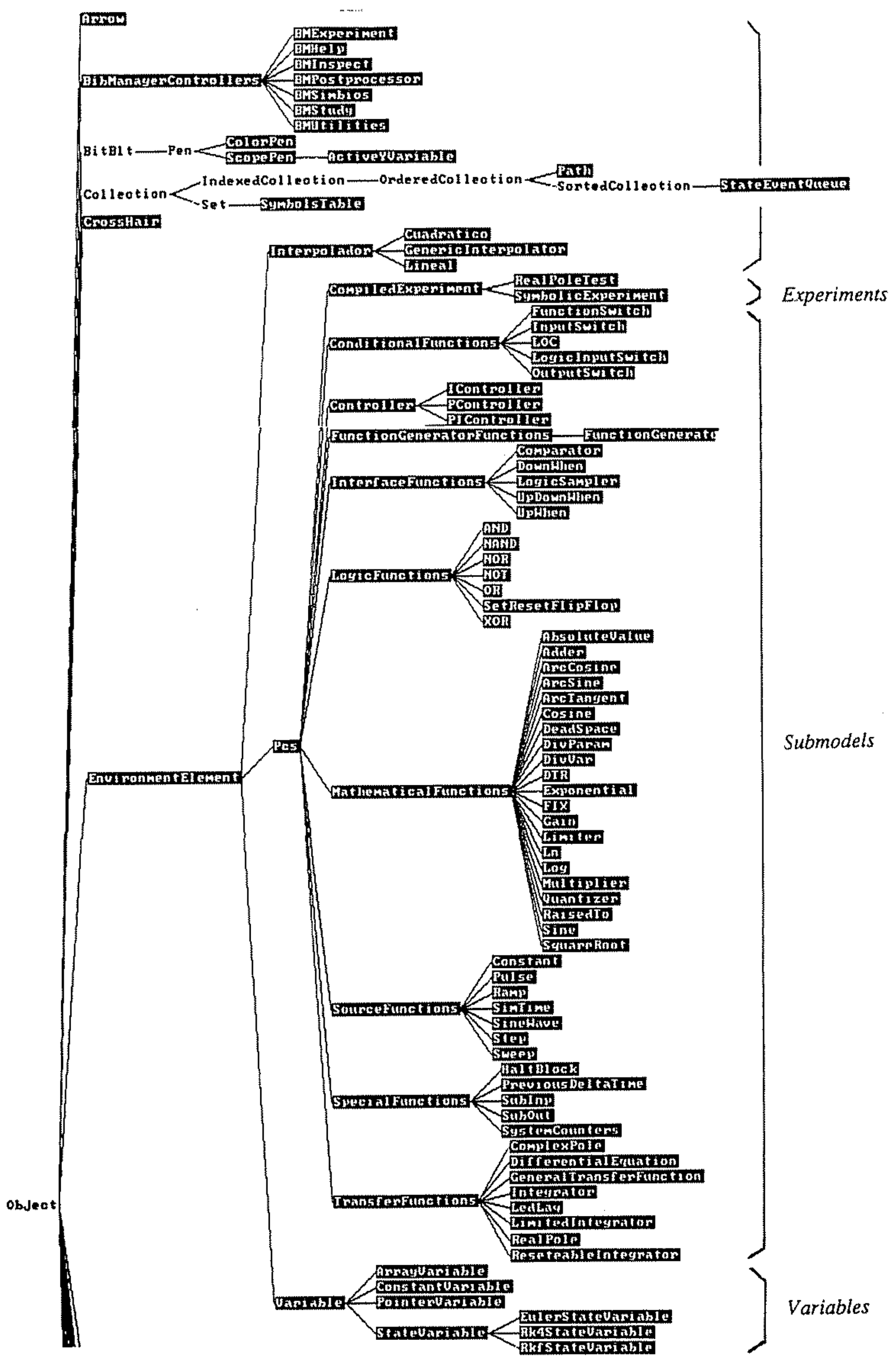


Figure 1. Model representation levels.


Object
Arrow
BibManagerControllers
BMExperiment
BMHelp
BMInspect
BMPostprocessor
BMSimbios
BMStudy
BMUtilities
BitBlt
Pen
ColorPen
ScopePen
ActiveYVariable
Collection
IndexedCollection
OrderedCollection
Path
SortedCollection
StateEventQueue
Set
SymbolsTable
CrossHair
EnvironmentElement
Interpolador
Cuadratico
GenericInterpolator
Lineal
Pcs
CompiledExperiment
RealPoleTest
SymbolicExperiment
ConditionalFunctions
FunctionSwitch
InputSwitch
LOC
LogicInputSwitch
OutputSwitch
Controller
IController
PController
PIController
FunctionGeneratorFunctions
FunctionGenerato
InterfaceFunctions
Comparator
DownWhen
LogicSampler
UpDownWhen
UpWhen
LogicFunctions
AND
NAND
NOR
NOT
OR
SetResetFlipFlop
XOR
MathematicalFunctions
AbsoluteValue
Adder
ArcCosine
ArcSine
ArcTangent
Cosine
DeadSpace
DivParam
DivVar
DTR
Exponential
FIX
Gain
Limiter
Ln
Log
Multiplier
Quantizer
RaisedTo
Sine
SquareRoot
SourceFunctions
Constant
Pulse
Ramp
SimTime
SineWave
Step
Sweep
SpecialFunctions
HaltBlock
PreviousDeltaTime
SubInp
SubOut
SystemCounters
TransferFunctions
ComplexPole
DifferentialEquation
GeneralTransferFunction
Integrator
LedLag
LimitedIntegrator
RealPole
ReseteableIntegrator
Variable
ArrayVariable
ConstantVariable
PointerVariable
StateVariable
EulerStateVariable
Rk4StateVariable
RkfStateVariable
Experiments
Submodels
Variables

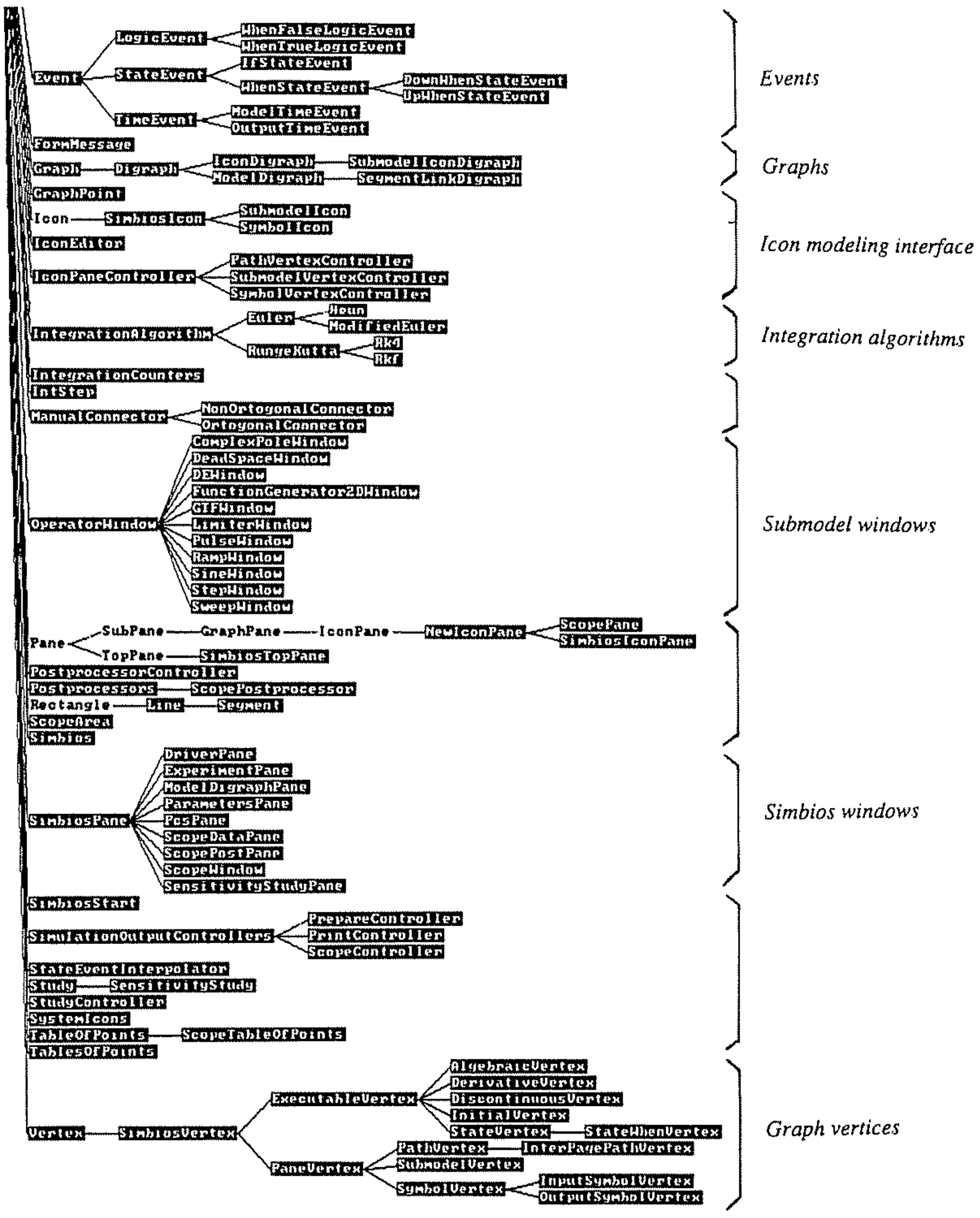


Figure 8. The SIMBIOS class hierarchy