Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

THE CONCEPT OF VIEWS IN SIMULATION

Margarita Rovira* David L. Spooner* Jorge Haddock**

*Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York 12180, U.S.A.

**Dept. of Decision Sciences and Engineering Systems
Rensselaer Polytechnic Institute
Troy, New York 12180, U.S.A.

ABSTRACT

The concept of views is widely used in the database community as a tool to reorganize and extend a database. It allows different users to look at the same data in different ways without changing the original data or violating its integrity. The same concept can be applied to a simulation model to find new high level interaction patterns among the components of a model without having to define new models and in a way that maintains consistency among the results. This paper makes a comparison of what the concept means in both contexts, describes a system that is under development to support views in simulation, and presents a small example.

1 CONCEPT OF VIEWS

The concept of views developed by the database community (Elmasri and Navathe 1989) allows the extension and reorganization of a database without changing the original data. Some of the benefits obtained from database views are that different users can look at the same data in different ways, data integrity is maintained among the different views because they are derived from the same source, and in most cases only the original data can be changed. The structure of a database view is of the same type as the original database. This allows the treatment of a view as a transparent extension of the original database since it is perceived as another component. The view is a virtual component though; it is usually not stored in physical storage and it is usually recomputed when referenced. A view is different from a real component of the database because its existence is dependent on the state of the other real components.

A view creates a new virtual component of the database from sections of one or more previously defined components. The previously defined components may be base components that hold the original data or other views. A view might include only sections of a previously defined component. This gives rise to another use of views: controlling access to database. A view may restrict the access to the database by making visible to a user only sections of the database that he/she is authorized to use.

One of the most difficult issues of database views is the updating of data within a view because there might not be an unambiguous mapping of the change to the components the data was derived from.

A database view is equivalent to the result of a query on the database. A query specifies the qualifications of a limited section of the database. The result of a query can be thought of as a collection of elements that meet the qualifications specified in the query. In terms of relational databases, the result of a query is a collection of tuples (not necessarily a set). Each tuple fulfills the qualifications of the query.

2 CONCEPT OF VIEWS IN SIMULATION

A view of a model in simulation allows the definition of a new component in the simulation that is derived from other components. These components can be sections of the original model or previously defined views. The overall objective is similar to the case of database views which is to allow the extension and reorganization of a model without affecting the original one. A simulation view is perceived as another component in the simulation with the same type of basic structure as the original components of the model from which it is derived.

A simulation view is an analysis tool that will allow the analyst to look for new insight into the high level interaction patterns of the components. In general, that is the objective of any model. We usually understand the low level interactions of the different components, which is what we represent in a model. The execution of that model gives us insight on the overall effect of the low level interactions on the system as a whole. A view extends the capacity to look for different patterns of interaction among the components of a model by allowing aggregation of components at different levels of abstraction based on different qualities. The expected result is more insight into the effects different qualities have on the overall system.

Another use of the view in simulation is for simplification. Complex systems require large models that usually take a long time to run. One of the objectives of supporting views is to have the ability to replace sections of the model with a simplified version of

its behavior that will accelerate the execution but will maintain a true representation of the real system. An example of a simplification is where a group of components is abstracted as a workcenter and instead of representing the behavior of the workcenter as a sequence of behaviors of its components it is replaced by a delay defined by a random variable. The behavior of the group of components is simplified by the use of the random variable. This type of use of a view allows the analysis of only sections of the model in detail, and allows the acceleration of each run.

3 DIFFERENCES BETWEEN DATABASE AND SIMULATION VIEWS

There are important differences between a database view and a simulation view. In the case of a database, a view might include just small sections of other components of the database. In the simulation case, a view includes whole components and may derive its state and behavior from the state and behavior of the components. The simulation view is an aggregation of lower level components that defines a new type of component.

Another difference is that in the database case a view reorganizes only data. In simulation a model is composed of a state, which is equivalent to the data of the database, and behavior. The simulation view has to abstract behavior as well as data for its components.

A view in simulation must be able to act as any other component of a model. The integration of a view into a simulation model may require changes in the interactions among the different components. The view facility in simulation becomes more complex than in the case of the database view since it has a more dynamic nature to it. A model in execution changes the state of its components during execution. Changes to the state of a view are an important issue.

A view in databases applies uniformly to all the contents of the database that meet the conditions defined by the query. In simulation a view may be required to hide only a section of the components that meet certain conditions. In other words, the simulation view must be applicable uniformly or non-uniformly to a model. For example, in simulation a view might abstract a queue and a machine as a new component called workcenter. The activation of the view does not necessarily mean that all queues and machines must be associated to represent workcenters, although the potential to do so exists.

The result of a query in relational databases is a multiset (sometimes called a bag) of tuples. Each tuple fulfills the conditions defined in the query. The group of tuples in the result of a query defines the content of the view. In simulation, a view will represent (using relational database terminology) a single tuple at a time. The result of a search in a simulation model for certain conditions might result in more than one disjoint set of components that fulfill the conditions. Each set would be represented by a view of the same type, but each is actually a different view. In other words, in databases there can only be one view of each type. Each view may contain more than one case that fulfills the conditions. In simulation there may be more than one view of the same type, each view encloses a single case that fulfills the conditions.

A database view is valid while the data it was derived from is not changed. As soon as any of the base data is changed the view is invalidated and needs to be recomputed if accessed. The database view is always active; once it is defined it can be accessed. A simulation view is only active during a run of the model where it is activated. The model must maintain its validity during the whole run by propagating the changes of the base components to the views that include those components.

4 SIMULATION ENVIRONMENT THAT SUPPORTS VIEWS

A simulation environment that supports the creation of views in an object-oriented environment is being built by the authors on top of an object system based on the Simulation Data Model (SDM), also developed by the authors. SDM is an extension of the traditional object model that promotes more reusability and extensibility.

4.1 Data Model.

A data model is an abstraction device that allows one to see the information content of data as opposed to the individual values of data. SDM is an object-oriented semantic data model that is based on three basic types of elements: model, behavior and cluster. SDM allows the hierarchical composition of any type of element, and allows the definition of modular elements that can have a delayed definition of their composition and interactions. SDM is implemented as an object system that introduces the three basic elements as metaclasses from which everything else is derived.

Zeigler (1990) defines a module as program text that can function as a self-contained autonomous unit in the following sense: interactions of a module with other modules can only occur through pre-declared input and output ports. Traditional objects and modular system models share the concept of internal state. Objects allow the hierarchical composition of state and the late binding of attributes to values, but they provide very little flexibility in the composition of behavior and the late binding of interactions among objects.

SDM is an extension of the object-oriented model that shares more features with the modular system model than the traditional object-oriented model. SDM is object-oriented and provides encapsulation, inheritance and message passing. It applies the encapsulation not only to models that are close to the traditional concept of an object, but also to behaviors, allowing behaviors to

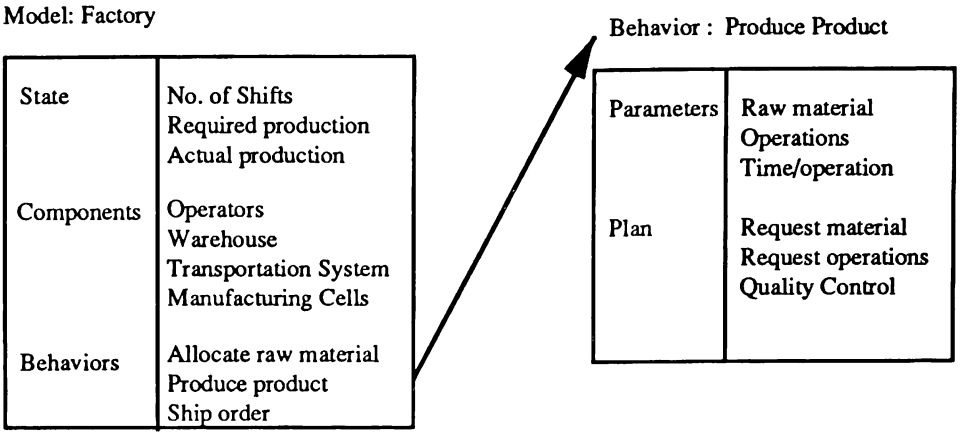

Figure 1. Model and Behavior Interrelationship

be composed dynamically from other behaviors and associated with a variety of models.

Models are objects that have a state, a group of subcomponent models, and a list of behaviors that are themselves objects. Each part of a model allows delayed binding, even the behaviors. A model has an attribute for each behavior that may act on it, and the value of this attribute is an instance of the appropriate behavior.

Each behavior is a separate object that includes a set of parameters and the description of its action. The description of an action may be an atomic function or a sequence of lower level behaviors that need to be activated according to some defined order when the behavior is executed. A behavior is activated by sending a message to it. The message can set the parameters of the behavior, customizing its action or its interaction with other behavior objects.

Figure 1 shows the interrelationship among model and behavior objects. The state of a model includes plain attributes that hold values, components that hold instances of lower level models, and behaviors that hold instances of behavior objects. Behaviors are separate objects that may exist independently. Only in the case when an instance of a behavior is held by a model can the behavior act on the model.

In addition to models and behaviors, the third type of element SDM supports as a first class object is the cluster. Clusters represent groupings of objects that share common properties. For example, a cluster might represent a group of machines that have failed more than three times. The definition of the properties of a cluster allow the creation of new clusters with members explicitly chosen from existing clusters or implicitly identified through characteristics the members share. The cluster is the basis for the support of views in a simulation environment.

SDM also allows the modeling of user-defined relations among the different types of objects. A relation in SDM allows a 1-to-many link between two types of objects. A relation may be associated with an inverse relation which also allows a 1-to-many link. The user may define any type of relation between any type of objects. Behaviors can be used to create links between objects as well.

Relations and views are powerful tools to gain insight on high level interaction patterns of a system.

4.2 Simulation Environment.

The simulation environment proposed for SDM is an object-oriented environment that separates model construction and experiment specification as promoted by Ören and Zeigler (1979, 1986).

Model construction is supported by three hierarchies of classes that can be re-used, extended and customized. Each hierarchy derives from one of the three basic elements of the data model: model, behavior and cluster.

The model hierarchy starts with a SimulationModel class that is a direct subtype of model, the root of the hierarchy. SimulationModel provides the basic functionality to be able to accumulate statistics for the different models. SimulationModel provides the capability to record two types of statistics: observational data and time-dependent data. Observational data is recorded as a sequence of equally weighted observations, where each observation is triggered by a user-activated behavior. Time-dependent data is recorded automatically by the system whenever the value of a user-defined dependent variable changes. The system keeps track of the value of each variable and the interval of time for which that value is valid.

Any type of simulation model is derived from SimulationModel so that it can keep track of statistics. The basic types of models defined are Entity, Resource and Queue. See Figure 2. An Entity object represents an active object that moves around the environment and has a temporary life span. A Resource object represents a passive object that has a more permanent nature and whose activity depends on requests from the active objects. The Queue object represents an ordered grouping of objects.

Subclassing of the basic models allows the user to customize the simulation environment to a specific application and to make it represent more closely the real world. That has always been one of the advantages of object-oriented languages for simulation.

The models being implemented as proof-of-concept of the benefits of views in simulation are based on a queuing model. That does not mean that the environment is restricted to this type of modeling. One of the motivations for defining a simulation environment from the ground up is to achieve greater flexibility in the type of modeling that can be supported.

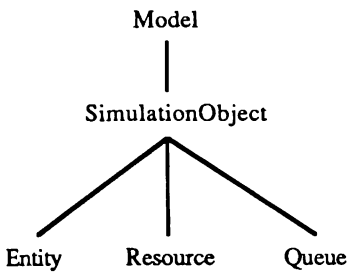

Figure 2. Model Hierarchy

The Behavior hierarchy defines a database of generic behavior that can be customized and shared by different types of models. See Figure 3. The separation of the behaviors from the models promotes sharing of common behavior among models that are not necessarily related by the is-a relationship. Traditional object-oriented systems only allow sharing behavior through inheritance within the class hierarchy. A generic behavior defines the binding constraints that must be met by a model to be able to use it. For example, a behavior might define as a binding constraint that the model includes an attribute with a certain name and of a certain type in order to allow it to be bound to the model.

The behavior hierarchy of the simulation environment provides very basic and common behaviors for queuing models. As we said before, this does not mean that it is restricted to only this kind of behavior. New behavior can be composed from the basic ones or any other pre-defined behaviors.

One of the major drawbacks of the object-oriented languages for simulation has been the required knowledge of the language in order to extend a given environment. The user had to learn the syntax of the underlying language to write methods to extend the functionality of any system. This environment exploits the extensibility advantages of an object-oriented environment and provides an interface that can customize the environment without writing code. Given a large set of primitives, a user can create an environment without low-level coding.

The Cluster hierarchy allows the definition of group types that support the use of views. A cluster is qualified by the type of objects it may enclose, the cardinality (how many of each type can be involved in the group) and the common properties of the group. The properties provide a generic mechanism to define membership in a cluster group and may be compared to the query language in a database environment.

The environment provides only the root of the Cluster hierarchy. This object includes the functionality to look for the objects that fulfill a set of properties and to maintain the members of a cluster.

Ören and Zeigler (1979, 1986) define the experimental frame as a way of reflecting the objectives one has in experimenting with a model. Ören and Zeigler define as components of an experimental frame:

- the initial conditions governing the conduct of the experiment
- the rules by which the experiment comes to an end
- the analysis of the results of the simulation runs.

The concept of the experiment specification is supported by another hierarchy of classes, the Simulation hierarchy. The root of this hierarchy provides all the execution control functionality required in a simulation environment. In other words, it takes care of the scheduling of activities and the control of the simulated time. Besides the basic execution control functionality, it provides mechanisms that customize the experiment, and it provides a way of defining views of a model.

The Simulation hierarchy provides a way to define the experimental frame associated with a model during a run. All experiments include a single object derived from the Simulation class to control the experiment. A model can be associated with more than one descendant of this Simulation class, which allows the experiment to evaluate the same model under different contexts.

The view mechanism in the environment is also controlled through a descendent of the Simulation class. The original model will be associated with a given simulation instance. A subclass of that instance may

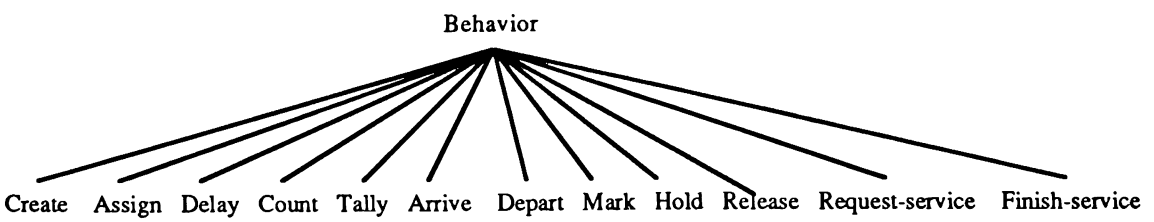

Figure 3. Behavior Hierarchy

create a view of the original model. Different views of the same model may be controlled by creating different subclasses of the instance that defines the original model. For example, the execution of the model of a manufacturing floor might be coordinated by a descendent of the Simulation class named ManufacturingFloor. If we want to concentrate on just the high level flow of all different parts, a view of the original model can be created by just creating a subclass of ManufacturingFloor. The original model is not changed at all. The view is activated as an extension of the original model. Any changes made to the original model will be inherited by the views, automatically assuring integrity and consistency among any number of views that derive from it. See Figure 4.

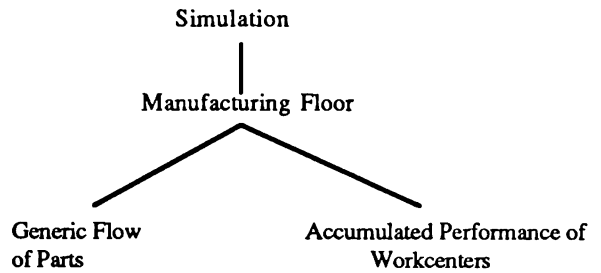

Figure 4. Simulation Hierarchy

4.3 Example

Let's take a very simple example to demonstrate the different concepts mentioned above. A manufacturing process for a part consists of one operation only. The part comes into the system, joins a queue if the machine is busy, gets operated on by the machine and then leaves the system.

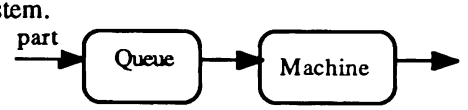

Figure 5. Original Manufacturing Process

The model that represents the above example will have to create an instance of a queue and of a resource to represent the queue and the machine respectively. Since the objects we are modeling do not exhibit any special behavior, we can just use the basic functionality of the classes provided by the system. So, our model starts by creating instances of the passive objects in the system and assigning them names that we can use to define the interactions among objects later. See Figure 6.

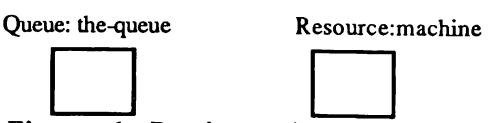

Figure 6. Passive object instances

The environment uses a process orientation view of simulation. The process is defined by detailing the life cycle of one particular part that goes through the system. The class Entity provides the basic functionality of the active object. This class must be customized with a behavior instance that describes the life cycle of the active object and is held in the attribute called lifecycle.

The first thing we need to do is to define a new type of behavior that will describe the process a part goes through while in the system.

Figure 7 shows the composition of a new behavior using existing behavior. It has four parameters that define its interaction with the outside simulation environment. The parameters of a behavior are used to define the inputs and outputs of the behavior, although they are not explicitly differentiated as such. In this case, the first two parameters are inputs and the last two outputs. The outputs indicate the objects that the behavior is supposed to interact with.

Behavior: part-process

parameters	arrival-time work-duration queue resource
Plan:	ask self arrive arrival-time ask queue hold self ask resource request ask queue release ask resource delay work-duration ask resource finish ask self depart

Figure 7. Process of a part

The plan defines the different steps of a composite behavior. The steps of a plan can be of two types: ask or tell. An ask step is similar to a procedure call, the execution of the plan will be blocked until the step is finished and a result received back. The tell step allows for concurrent execution of actions. The system does not wait for the ending of the tell step before the next step is executed.

In this example the actions are all sequential. The value of self within any behavior refers to the model that owns the behavior at execution time. The first step asks the owner of the part-process behavior to schedule an arrival of the entity. Arrive is a behavior of Entity and it expects one parameter. The arrival-interval parameter value is passed as a parameter to arrive.

The second step indicates that the object that is bound to the parameter queue is sent a message to execute the behavior hold. In this case the behavior hold is passed the value self, which places the owner of the part-process behavior, a part, on the queue. The third step is to send a message to the object that is bound to the parameter resource to execute the behavior request, whose objective is to get the resource assigned

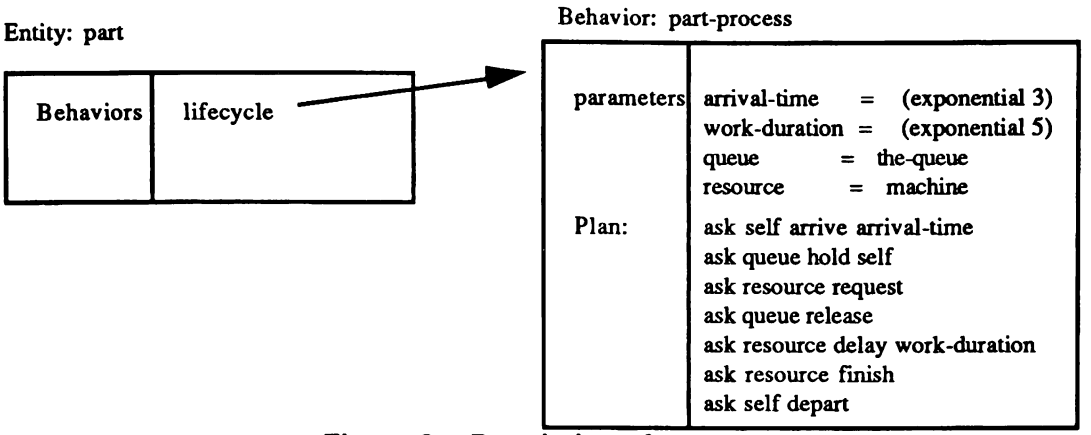

Figure 8. Description of a part

exclusively to the part. In this case no parameters are passed through the message. As soon as the resource is assigned to service the part, the next step starts executing, and the queue is sent a message to release the part. The duration of the service is simulated by a delay behavior that will block the execution for a time defined in the parameter work-duration. When the delay ends, the next step releases the resource. The last step of the behavior models the exit of the part from the manufacturing floor.

The next step in the modeling is to define a subclass of entity that behaves like the part and that interacts with the queue and the machine previously instantiated. See Figure 8.

The class part inherits the state from the class Entity, but redefines the behavior lifecycle to be of the type part-process. When a part is instantiated, the slot lifecycle will hold an instance of the behavior part-process. That part-process instance will customize its parameters by setting two of them to random numbers that have an exponential distribution and queue and resource to instances the-queue and machine, respectively. An instance of Entity must always have a behavior lifecycle initialized.

So far, we have constructed the model of the system and have defined the interactions among the different components. In order to run it we must define a subclass of Simulation that will hold the general description of the system and will define the control conditions of the execution.

Simulation: Manufacturing Process

active objects	part
final time	100
warm-up time	20
replicates	3
re-initialize system	yes

Figure 9. Subclass of Simulation

The manufacturing process instance defines the context of the model. See Figure 9. The simulation gets started by sending a message to the manufacturing process. The attribute active objects is a list of classes that will be activated at the start of the simulation. Each class is activated by creating its first instance and sending it a message to start its lifecycle.

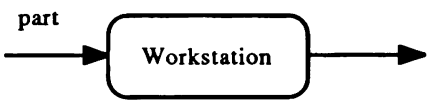

Figure 10. A view of the manufacturing process

Suppose now that we want to create a view of the model defined by manufacturing process. See Figure 10. We need to do the following:

A view is an object that will have the characteristics of a model and a cluster by inheriting from those two different hierarchies. Workstation is defined as a view that will have members of the type queue and machine. See Figure 11. When no properties are defined for a view the members are explicitly defined by the user. If properties are defined the members are determined automatically by the system based on the properties.

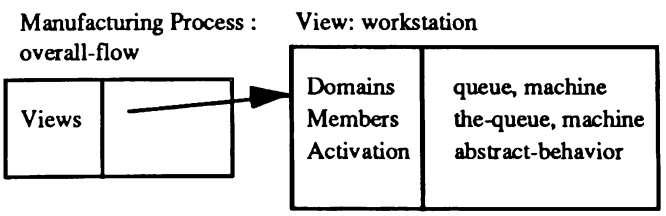

Figure 11. Creating a view of Manufacturing Process

Overall-flow will inherit all the characteristics of manufacturing process. See Figure 11. Namely, the

original model gets activated and any other control descriptions of the experiment that are not redefined by the subclass. In this case we want to maintain all the control information but we want to create a higher level description of the system where the part only interacts with one object during its stay in the system, a workstation.

The attribute views of overall-flow can have a list of objects that will be activated as views. Each one can be customized and activated differently. A view can be activated in two ways: as a structural view or as a behavioral one. A structural view serves as a summary of the state of its members. In a behavioral view, the activation will create new behavior and will modify the life cycles of the entities that interact with the new object.

Running the model under overall-flow introduces the new component workstation as an active part of the simulation. This component hides the original components the-queue and machine. The process that describes the life cycle of the part must now interact with the workstation instead of the original components.

Behavior: part-process

parameters	arrival-time work-duration queue resource workstation
Plan:	ask self arrive arrival-time ask workstation process-part self work-duration queue resource ask self depart

Behavior: process-part

Parameters:	part work-duration queue machine
Plan	ask queue hold part ask resource request ask queue release ask resource delay work-duration ask resource finish

Figure 12. Generated Behavior

All the changes in the interaction among components can be generated automatically by the system if no simplification is required. This is possible because the interactions have been explicitly defined and can be changed. The generated behavior of the workstation, process-part, is defined as a sequence of steps that involve behaviors of the-queue and machine. The lifecycle of the part now interacts only with the workstation and passes the original parameters to the process-part behavior.

In the case of simplification, the user can replace the behavior process-part with a different one that simplifies the execution.

5 RELATED WORK

The concept of modular hierarchical systems defined by Zeigler(1990) has been coupled to object-oriented systems by several research groups (Pegden 1992). Hierarchical models allow the creation of models at different levels of abstraction by coupling together pre-existing modules. Each description of the same system at a different level of abstraction is a view of the system. However, the hierarchical model approach only addresses static changes in the level of abstraction of a model. Views of a system are completely independent models that must be maintained separately. The hierarchical model approach requires redoing a lot of modeling work and does not assure consistency among the views.

Rothenberg (1989) has also identified the need of a view capability in simulation. The concept of view in his case applies more to the world view used in the simulation (object-oriented view or event view of the same system). The concept of view, as used in this paper, is more consistent with what he calls different levels of aggregation of the same model.

6 CONCLUSION AND FUTURE WORK

Restructuring of patterns of interaction can provide great insight into dependencies of a system that might not be obvious at first sight. Rewriting of a model to restructure it is not a trivial problem, however. If the restructuring can not be done without having to redo existing work, it is more than likely that it will not be used as an analysis tool. The concept of views in simulation provides a mechanism to restructure models with very little effort and maintains consistency among the models. Hence, views have the potential to offer significant benefit to simulation.

Objects in an object-oriented environment present a very natural representation of views (Abiteboul and Bonner 1991) (Widerhold 1986). Unfortunately, traditional object-oriented languages present limitations for the integration of a view into a simulation because interactions among objects cannot be changed at execution time. The approach taken in this project has been to start from the ground up, extending the object system that supports the simulation environment in order to support views. The resulting system, besides making the support of views possible, presents potential for the development of more intelligent tools to support simulation.

More work remains to be done for views. The approach that is under evaluation defines views that last throughout the execution of a model. Another approach to be evaluated is the activation of views in real-time, where the execution of a model can be suspended, a view activated, and execution resumed.

Views create higher level abstractions from an existing model. It is also important to investigate providing the capability to move to a lower level of abstraction from the original model, so that components can be decomposed into lower level components integrated into the execution of the model.

ACKNOWLEDGMENTS

This work has been partly supported by the National Science Foundation under Grant No. DDM-9200517.

REFERENCES

Abiteboul, S. and A. Bonner. 1991. Objects and Views. Proceedings of the 1991 SIGMOD Conference.

Cammarata, S. J. and C. Burdorf. 1991. PSE: an object-oriented simulation supporting persistence. Journal of Object-Oriented Programming, Volume 4, Number 6, pp. 30-40.

Elmasri, R., and S. B. Navathe. 1989. Fundamentals of Database Systems. The Benjamin/Cummings Publishing Company, Inc. Redwood, California.

Elzas, M. S., T. I. Ören and B. P. Zeigler (Editors). 1986. Modeling and Simulation Methodology in the Artificial Intelligence Era. Elsevier Science Publishers B.V. (North-Holland).

Pegden, C. D. and D. A. Davis. 1992. Arena: A SIMAN/Cinema-Based Hierarchical Modeling System. Proceedings of the 1992 Winter Simulation Conference. Arlington, Virginia.

Rovira, M., J. Haddock, and D. L. Spooner. 1993. Dynamic Abstraction Mechanisms for Simulation Environments. Proceedings of the 1993 NSF Design and Manufacturing Systems Grantees Conference, University of North Carolina at Charlotte, Charlotte, N.C.

Rothenberg, J., S. Narain, R. Steeb, C. Hefley and N. Shapiro. 1989. Knowledge-Based Simulation: An Interim Report. A Rand Note, N-2897-DARPA.

Wiederhold, G. 1986. Views, Objects and Databases. Computer. December 1986. Volume 19 Number 12.

Zeigler, B. P. 1990. Object-oriented Simulation with Hierarchical, Modular Models: intelligent agents and endomorphic systems. Academic Press Inc.

Zeigler, B. P., M. S. Elzas, G. J. Klir and T. I. Ören (Editors). 1979. Methodology in Systems Modeling and Simulation. North-Holland Publishing.

AUTHOR BIOGRAPHIES

MARGARITA ROVIRA received a B.S. degree in computer science and a M.S. in industrial engineering from the Georgia Institute of Technology. She received a M.S. degree in computer science and is currently working toward the Ph.D. degree in computer science at Rensselaer Polytechnic Institute. Her main research topic is the development of generic tools to support dynamic changes of abstraction in simulation environments. Other research interests include data modeling, object-oriented technology and databases.

DAVID L. SPOONER received his PhD degree from the Pennsylvania State University in Computer Science. Since then he has been at Rensselaer Polytechnic Institute in Troy, New York, where he is a Professor in the Computer Science Department. He is a co-director of the Data Engineering Program in the Rensselaer Design and Manufacturing Institute. His research interests include engineering database systems, object-oriented systems, and database security.

JORGE HADDOCK is an associate professor of Industrial Engineering and Operations Research in the Department of Decision Sciences and Engineering Systems at Rensselaer Polytechnic Institute. He holds a BSCE from the University of Puerto Rico, a MSMgtE from Rensselaer, and a PhD in Industrial Engineering from Purdue University. His primary research interests involve modeling of manufacturing / production and inventory control systems, as well as the design and implementation of simulation modeling and analysis tools.