OBJECT–ORIENTED MODELING: TOOLS AND TECHNIQUES FOR CAPTURING PROPERTIES OF PHYSICAL SYSTEMS IN COMPUTER CODE [†]

François E. Cellier and Bernard P. Zeigler
Dept. of Electrical & Computer Engr.
The University of Arizona
Tucson, Arizona 85721
U.S.A.

Andrew H. Cutler
Space Engineering Research Center
The University of Arizona
Tucson, Arizona 85721
U.S.A.

Abstract. Mathematical modeling means the formal encoding of knowledge about a dynamical system. Knowledge can be grouped into behavioral knowledge and structural knowledge. Behavioral knowledge is local knowledge relating to a particular experiment applied to a system or model. Behavioral knowledge is what is generated in a real–world experiment or during a simulation run. Structural knowledge is global knowledge relating to a system or model, irrespective of the experiment that is performed. A model is a formal encoding of structural knowledge of a system. Structural knowledge can be further decomposed into functional knowledge, coupling knowledge, decomposition knowledge, and taxonomic knowledge. In this paper, a methodology is presented that helps to organize the structural knowledge about a system to be described. It enables to encode separately and in an organized fashion functional, coupling, decomposition, and taxonomic knowledge about a system. The methodology lends itself to the implementation of automated procedures for deductive as well as inductive model synthesis necessary for the realization of high–autonomy intelligent control systems. A fairly involved example concludes the paper.

Keywords. Artificial intelligence; decentralized control; event–based control; failure detection; intelligent machines; knowledge abstraction; model synthesis; object–oriented modeling; process control; time–windows.

INTRODUCTION

New buzz words appear regularly in the software engineering literature. One such buzz word of a fairly recent vintage is *object orientation* (Booch, 1991). The idea of object orientation was first popularized by computer languages, such as Smalltalk (Goldberg and Robson, 1983) and by operating environments, such as the Hypercard (Shafer, 1988). Suddenly, any piece of code had to be "object–oriented" in order to be respectable.

After a certain delay time most software engineering buzz words show up in the world of simulation, so also the term "object orientation" (Zeigler, 1990). Consequently, simulation software suddenly had to be object–oriented. What is object–oriented simulation? Before this question can be answered, it is necessary to take a step back and discuss the basic idea behind object orientation in general. What is object orientation? Where does the demand for object orientation come from? Why is object orientation a useful attribute of a software system? These are some of the question that should be answered before the concept is blindly ported to a particular application area such as simulation.

What is object orientation? Stated in simple terms, object orientation of a piece of software can be defined as its capability to represent facets of real–world objects in a compact and coherent form. Figure 1 shows a few "real–world objects" (modestly abstracted) that are partially embedded inside each other.

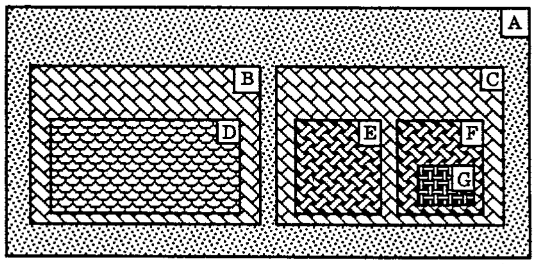

Figure 1. A set of "real–world objects."

Object A is the "world object," i.e., an object that encompasses the "entire world" (as far as it is of concern to the task at hand). The world object contains two embedded objects, B and C, and as far as A is concerned, these two objects are the only objects in the whole world. Object B contains one object, D, while object C contains two objects, E and F. Finally, object F contains object G. Object C knows that it is not the entire world. The traditionally employed view was the following: object C knows that it is embedded in object A, but it doesn't know whether A is the world object or whether A is further embedded in a yet more comprehensive object. Object C knows about its own two children objects, E and F, but it doesn't know

[†] This research is supported in part by NASA-Ames Cooperative Agreement No. NCC 2-525, "A Simulation Environment for Laboratory Management by Robot Organizations," and in part by the NASA/University of Arizona Space Engineering Research Center for the Utilization of Local Planetary Resources (SERC/CULPR).

anything about its grandchild, G, and it doesn't know anything about its sibling, B. This is the so-called *hierarchical world view*. A number of computer languages were based on this world view. The most prominent among them is Pascal. In Pascal, the world object is called the *program*, whereas embedded objects are called *procedures*. In Pascal, the world of Fig. 1 can be represented as follows:

```
program A
    procedure B
        procedure D
            ...
        end
        ...
    end
    procedure C
        procedure E
            ...
        end
        procedure F
            procedure G
                ...
            end
            ...
        end
        ...
    end
    ...
end .
```

Graphically, a strict hierarchy can be represented as a so-called *system entity structure (SES)*. Figure 2 shows the SES of the previously discussed world.

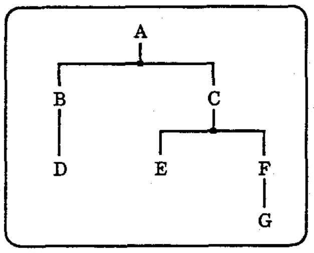

Figure 2. System entity structure of a "world."

The nodes of the SES are called *entities*. The world object is here called the *root entity*, while objects that are not further decomposed into other objects (in the preceding example: D, E, and G) are called the *leaf entities*. Leaf entities represent so-called *atomic objects*, whereas all other entities represent *molecular objects*.

The strict hierarchy has a number of awkward properties that deserve to be mentioned.

(1) Objects D and G could be twin siblings, i.e., they could be two instantiations of the same object type. Employing a strictly hierarchical world view, these two objects must be manually duplicated in the code. In fact, the strict hierarchy doesn't support the concept of object types (usually called object classes), but only individual objects.

(2) One of the demands of object-oriented programming is that things that belong together are placed close to each other. The strict hierarchy certainly does not provide for this feature. In Pascal, the variables of the world object (the program) are declared at the very beginning of the program, whereas the instructions performed on these variables are coded at the very end. Consequently, the Pascal programmer must constantly flip back and forth through many pages of code.

(3) One of the demands of object-oriented programming is that modifications that concern only one object can be handled locally. The strict hierarchy does not provide for this feature either. If one of the objects (procedures) is slightly modified, the entire program must be recompiled.

Of course, many "modern" Pascal compilers provide partial answers to all of the above problems. They achieve this by violating the rules of the strict hierarchy.

What are the major causes for the aforementioned problems? It turns out that the most serious deficiency stems from the demand that siblings shouldn't know anything about each other. In the real world, each human interacts constantly and directly with all the other "objects" in his or her environment. In fact, the human has only a very limited understanding of the "world object" and the "children objects" (such as the functioning of the brain), while he or she has a much better understanding of sibling objects. This idea has been ported to the *object-oriented world view*. The object-oriented world is populated by individual objects that "talk" to each other or rather that ask each other questions. In the object-oriented jargon this is called "asking a method from another object" — a fairly strange and somewhat confusing terminology.

The major paradigms of the object-oriented world view are the following:

(1) Objects are instantiations of generic objects, called object classes. An object class declaration is similar in nature to a Pascal **type** declaration, whereas an object declaration is similar to a Pascal **var** declaration.

(2) Each object possesses a knowledge base, i.e., a number of things that it knows and can do. These items are called *methods* and are coded as algorithms inside the object.

(3) Objects automatically inherit all properties (methods) of their respective class declarations.

(4) An object class can inherit methods of its parent class.

(5) Objects know about the methods of their siblings. An object can "ask a method from another object," i.e., it can ask the other object to run one of its algorithms and return the result of that algorithm. The Sir object asks the clock object "What's the time"? The clock object executes the algorithm that computes the current time, and replies: "It's ten o'clock, Sir."

(6) Implementational details about the methods contained in an object are hidden from the outside. The object-oriented world view provides for clean interfaces with few side effects.

The object-oriented world view has indeed overcome most of the problems inherent in strict hierarchies. Individual object classes can be coded and debugged separately with little to no side effects noticeable in other object classes. This enables a team of programmers to join in the efforts of coding a large software system (a so-called *case* — the youngest animal in the zoo of software engineering gibberish).

It should be noticed that objects are not hierarchically structured in the standard object-oriented world view, only classes are. However, even class hierarchies are somewhat different from programming hierarchies in more traditional approaches to programming. A subclass may inherit all methods of its parent class and add a few methods of its own. Thus, a subclass usually provides a *superset* of methods (capabilities) in comparison with its parent class, not a *subset*.

Yet, new problems popped up, which interestingly enough are again related to difficulties with hierarchies. The inconspicuous statement that "an object class can inherit methods of its parent class" puts the skeleton right back in the closet. By stating that all object classes (except the world object class) have a parent class, a hierarchical decomposition of classes is implicitly introduced, which causes the same problems as before. A *trap door* object should be allowed to inherit methods of its father, the *trap*, and its mother, the *door*. Unfortunately, strict hierarchies don't understand the concept of two parents. Some object–oriented programming environments provide for multiple inheritence schemes by allowing declarations such as:

class *trap_door* **is a** *trap* **and a** *door*

thereby again violating the concept of a strict hierarchy (Davis *et al.*, 1990).

The second problem has to do with the idea of "talking." The object–oriented world view presumes that all interactions between objects can be realistically represented by mechanisms of message passing. This is not so. If a drunkard drives his car against a tree, a strong interaction between the car object and the tree object takes place, which cannot properly be modeled as a mechanism of talking. The "talking" world view assumes that every interaction between objects can be described by one object *causing* an *effect* in another object. However, the cause–and–effect relationship is not always realistic. Does the tree cause the dent in the car? The tree hasn't done a thing! Is it the potential difference between the two ends of a resistor that causes a current to flow, or is it the current flowing through the resistor that causes a potential drop? In a mechanical system consisting of a *mass* object series–connected with a *spring* object, who owns the reaction force between the two objects?

MODELING VERSUS SIMULATION

Before these very interesting questions, which are intimately related to the topic of modeling, can be answered, it is necessary to properly distinguish between the process of modeling and that of simulation.

Modeling denotes the process of creating a model. *Simulation* describes the method of extracting trajectory behavior form a given model. In the past, these two terms have often been confused. There was no need for a clear distinction since modeling was an activity of the human user whereas simulation was performed by a computer program. However, it will be shown that there is good reason to automate the process of modeling as well. Consequently, a modern modeling and simulation environment (a modeling/simulation *case*?) should provide for both modeling and simulation software.

In such an environment, the human user interacts with *modeling software* that helps him to create a model. Once the model has been created it can be transformed (compiled) into code that can be understood by the *simulation software*. Thus, the so–called simulation program is now totally machine–generated and there is no longer any need for the human user to even understand the simulation language. The simulation program now assumes a role similar to the compiled machine–code program in an ordinary programming environment. Most human users of Pascal are unable to understand the code that is generated by the Pascal compiler and, hopefully, they never have a need for such knowledge.

In this context, it makes sense to distinguish between the terms *object–oriented modeling* and *object–oriented simulation*. The former term denotes the concept of an object–oriented user interface. The user models his system through objects and relations between these objects. The latter term implies that objects communicate by means of message passing mechanisms during the execution of a simulation run. These two properties have little in common with each other. It turns out that the modeling compiler (i.e., the program that generates the simulation program from the user–specified model) is perfectly capable of doing away with object orientation and translating an object–oriented model into an amorphic simulation program. For most purposes, that is exactly what the modeling compiler should do, i.e., while *object–oriented simulation* (as will be shown) is a fairly dubious undertaking, *object–oriented modeling* is a fruitful concept.

Ordinary object–oriented languages could employ the same concept, i.e., the object orientation could be put to the sword in the process of compilation. This is not usually done, because it would make it very difficult to still provide for separate compilation of objects. In ordinary programming, the compile time of a program is usually of the same order of magnitude as its execution time. Thus, efficient compilation is very important and separate compilation of subprograms (modules, objects) is therefore a very desirable feature. In simulation, however, the average execution time of the once–compiled simulation program is usually hundreds of times slower than its compilation. Consequently, efficient compilation of a simulation program is much less important than efficient run–time execution. For these reasons, the capability to separately compile simulation objects is not as high on the feature wish list and may be sacrificed if the price to be paid would be a considerably slower run–time execution.

OBJECT–ORIENTED SIMULATION

Object orientation in ordinary programming was introduced as a means to organize knowledge about the problem to be solved. Consequently, object orientation is important at the user interface, not at execution time. There is no run–time advantage to object orientation, except in a multiprocessor environment in which different objects are implemented on separate CPUs.

Since simulation programs (in the proposed modeling/simulation environment) aren't handcoded, there is no advantage to object–oriented simulation, except if the system to be simulated is a physically distributed system in which the physically distributed objects communicate by means of message passing. Typical examples include the simulation of a multiprocessor computer architecture, the simulation of a national communication network, or the simulation of the distributed control architecture of a large energy distribution network. Zeigler's book presents many meaningful examples of the use of object–oriented distributed discrete–event simulation (Zeigler, 1990).

Object–oriented discrete–event simulation is not necessarily desirable, but at least it comes about quite naturally. Objects in a discrete–event system communicate (by definition) through mechanisms of message passing, i.e., there exists a natural cause–and–effect relationship between all objects in a discrete–event system. There is never any question as to which object is the cause of an event and which objects are affected by it.

Object–oriented continuous–time simulation is much more problematic. Since continuous objects "communicate" continuously with each other, the "talking" paradigm may not be such a useful concept altogether. Chicken–and–egg type conflicts (such as in the previously mentioned case of the voltage across and the current through an electrical resistor) occur frequently and interfaces between objects often introduce "methods" of which it is unclear to whom they belong (such as the reaction force in the aforementioned mass–spring system). Furthermore, due to the needs for continuous (or at least very frequent) communication between different continuous objects, the run-time efficiency issue becomes a real bottleneck. There is simply no efficient way to implement two continuous real-world objects (such as the aforementioned mass and spring objects) as two separate software objects and yet have them communicate efficiently by exchanging information at least once per integration step (not once per communication interval). For these reasons, object-oriented continuous-time simulation is a nono!

Interestingly, the object-oriented paradigm is much older than either Smalltalk or Hypercard. It was originally proposed in the context of discrete-event simulation. SIMULA'67 had all the properties of a modern object-oriented programming environment. It is sometimes difficult to understand why particular languages become fashionable while others don't. In the case of SIMULA, the language was much more promising than its success seems to indicate. The reason for this discrepancy is probably related to poor salesmanship. SIMULA'67 was advocated as a discrete-event simulation language — which it wasn't, since it didn't offer proper support for random variate generation, queue management, and output presentation. Correspondingly, a model that could be coded in five lines using GPSS called for a fairly long SIMULA program. SIMULA'67 was a quite decent language for compiler writing, but it wasn't properly marketed for that purpose.

OBJECT-ORIENTED MODELING

Object-oriented modeling helps the (human) user to organize his or her knowledge about the system to be modeled. All physical properties of one object should be expressible in one place.

A hierarchical decomposition of complex objects into simpler objects is desirable since each object, including the world object, should be expressible in terms of no more than 50 lines of code (N. Wirth, personal communication). This demand is incoherent with the standard object-oriented approach according to which all objects are at the same level. However, the object-oriented world view can easily be extended to encompass hierarchies of objects. Each method is an algorithm, i.e., a program. If a method is sufficiently complex, the program implementing that method can become very long. Consequently, it may be desirable to implement individual methods using the object-oriented world view. Thereby, a method inside an object is implemented by means of many objects generated from their own object classes that cooperate in task solving.

For reasons that will be explained in due course, it is important that all equations describing the system behavior are captured in *atomic objects*. A *molecular object* consists only of declarations of all the objects that it contains and a description of the interactions between them. This recipe will be referred to as the *coupling rule*. The description of an atomic object is called an *atomic model* and the description of a molecular object is called a *coupling model*.

In the preceding example, the leaf entities D, E, and G describe three physical objects. All other objects describe how these three physical objects are connected together to form a whole. Of course, the coupling rule precludes molecular objects from containing only one object. For example, the object F is not meaningful since it contains only one object G, and unless F contains physical equations besides a declaration of G, this makes little sense since F and G would be identical objects.

It should be noticed that only some of the properties of object-oriented programming have been migrated to the object-oriented modeling paradigm. In particular, the nature of connections is not specified at the modeling level. If the model represents a discrete-event system, the connections will be transformed into some sort of communication protocol during the transformation of the model into a simulation program. If the model represents a continuous-time system, the interactions between objects describe their physical couplings. In the mass-spring example, the mass and the spring are two atomic objects, whereas the mass-spring model is a coupling model that describes how the mass is attached to the spring. During the transformation, the coupling equations (possibly involving the aforementioned reaction force) are automatically generated from the declared nature of the coupling.

A SPICE program is an object-oriented description of an electrical circuit. The SPICE "models" (i.e., the elements R, C, Q, etc.) are the atomic objects. Subcircuits are the coupling models. They contain references to atomic objects and other subcircuits, and once-defined, they are molecular objects (X elements) that can be invoked just like atomic objects. The SPICE program is the world object.

AUTOMATED MODEL SYNTHESIS

A modeling/simulation environment was created, which comprises a five-level hierarchy of modeling tools (Cellier *et al.*, 1990; Cellier, 1991). The five-level hierarchy is shown in Fig. 3.

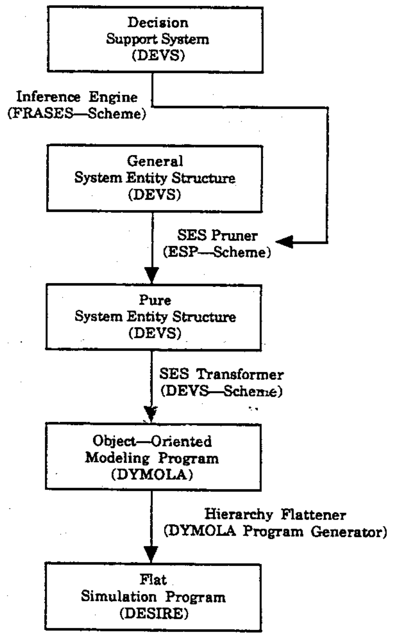

Figure 3. Five-level hierarchy of model management.

Flat simulation models are at the lowest level of the hierarchy. Simulation models can be either of the discrete-event type (coded in DEVS-Scheme (Zeigler, 1987, 1990)) or of the continuous system type (coded in either DESIRE (Korn, 1989) or ACSL (MGA, 1986)). As explained earlier, at least the continuous-time models must be flat in order to run efficiently. Unfortunately, flat simulation models are difficult to construct, read, and maintain. However, since these simulation models are not handcoded, this is not much of a problem. Level 1 models are optimized for run-time efficiency.

Hierarchically decomposed modular models of both types are at the next higher level of the hierarchy. Discrete-event hierarchical models are coded in DEVS-Scheme, while continuous–time hierarchical models are coded in DYMOLA (Elmqvist, 1978). Level 2 models are easier to maintain since they can reference submodels that can be stored in a model library. Submodels used in a level 2 model are instantiations of model types. Level 2 has been optimized for user convenience in coding atomic models to be stored in a level 2 model library. The transformation of level 2 models into level 1 models is accomplished by a *hierarchy interpreter*. In the case of discrete–event models, the hierarchy interpreter is a built–in function of the DEVS–Scheme simulation engine, whereas in the continuous case, the hierarchy is flattened by the DYMOLA preprocessor. The continuous case is a little more difficult to handle since continuous models do not provide for a natural distinction between component inputs and outputs. For example, an electrical resistor calls for the model:

$$U = R * I \qquad (1a)$$

when connected to a current source, but it requires the model:

$$I = U/R \qquad (1b)$$

when connected to a voltage source. The DYMOLA preprocessor contains formula manipulation algorithms that enable it to solve equations at compile time for the appropriate variable.

At the next higher level, models are represented by a pure system entity structure (SES) (Rozenblit *et al.*, 1989; Zhang and Zeigler, 1989). A pure system entity structure is a hierarchical tree that decomposes root entities (corresponding to the main program) graphically into its component models. The leaves of the tree correspond to level 2 atomic models, whereas the interior nodes of the tree correspond to level 2 coupling models. DEVS–Scheme contains a function (transform) that compiles a pure SES into a set of level 2 models. Atomic models are retrieved from the model library, while coupling models are automatically being generated from the information provided in the pure SES. The transform routine can generate level 2 models of either the discrete–event type (coded in DEVS–Scheme) or the continuous system type (coded in DYMOLA), and it will generate coupling models in accordance with their use. Thus, while atomic models are handcoded at the second hierarchy level, the coupling models are automatically generated from a higher level description. The third hierarchy level has been optimized for user convenience in specifying the coupling relationships between objects (the *coupling knowledge*). The coupling knowledge is specified by means of *stylized block diagrams*. In the system entity structure, each decomposition of an object into parts is represented by a so–called *aspect node*, i.e., layers of entity nodes toggle with layers of aspect nodes. The aspect nodes were omitted from the SES in Fig. 2. Each aspect node is linked to a stylized block diagram that encodes the coupling knowledge, i.e., shows how the parts are connected to form the object. "Signal paths" in the stylized block diagram represent (groups of) variables of both the *across* type (i.e., variables that assume the same values around a node, such as potentials in an electrical circuit) and the *through* type (i.e., variables that add up to zero in a node, such as currents in an electrical circuit). Unlike the conventional block diagram, the stylized block diagram does not mark the direction of the signal flow from one block to another. Thus, signal paths are undirected. This rule reflects the previously mentioned difficulty as expressed in Eqs. (1*a*–*b*). In the case of physical systems, an alternative to the stylized block diagram is the *bond graph* (Cellier, 1990). It can also be used to encode the coupling knowledge of a physical system.

At the next higher level, models are represented by general system entity structures. General SESs provide for a mechanism to describe many *variants* within one single SES. DEVS–Scheme provides for a tool, called ESP–Scheme (Kim, 1988), which can prune a general SES to generate a pure SES. In the pruning process, all variants except one are pruned out by cutting away all undesired branches of the SES. By selecting an internal node of the general SES as the root node of the pure SES, the pruner can also generate a pure SES of a subsystem only. Level 4 models are much more compact than level 3 models, since an entire class of models can be represented with one single general SES. Models at this hierarchy level have been optimized for specifying the *decomposition knowledge* of a system. The decomposition knowledge is encoded in the general SES, which is user–coded at the fourth hierarchy level. Entities can be decomposed into parts by use of aspect nodes, as explained earlier. They can also be decomposed into variants by means of so–called *specialization nodes*. Thus, in the general SES, layers of entity nodes toggle with layers that contain both aspect nodes and specialization nodes. Aspect nodes represent and connections: a car consists of a body, and an engine, and four wheels, and a trunk. Specialization nodes represent or connections: an engine can be either a V6 or a Diesel engine. Figure 4 shows the general SES encoding this example.

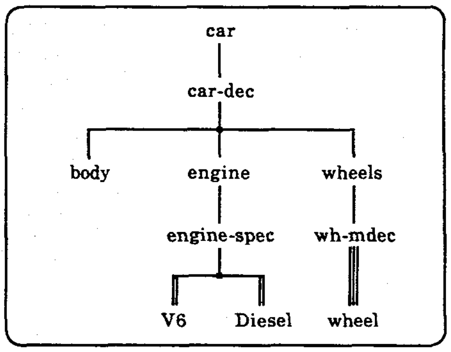

Figure 4. General (partial) SES of an automobile.

The only difficulty left is the need to decide manually which variant to choose among the many possible variants contained in a general SES. The fifth level of the hierarchy defines a rule–based decision support system that, on the basis of qualitative rules, instructs the pruner which branches to cut. This tool is called FRASES–Scheme (Frames and Rules Associated System Entity Structure) (Hu *et al.*, 1989). The fifth hierarchy level has been optimized to encode the *taxonomic knowledge* of the system. For this purpose, each specialization node is linked to a rule–base editor in which selection rules can be specified, such as:

```
if purchase_price = high and maintenance_cost = low
    then select Diesel
    else select V6
end if
```

A complete description of a taxonomy of models together with a complete description of a taxonomy of experiments to be performed on these models, i.e., a so–called *world model* consists of:

(1) a library of level 2 leaf models and level 2 unit action plans, encoding the functional knowledge of system components;

(2) a set of level 3 stylized block diagrams describing the coupling mechanisms between submodels of a decom-

position and tasks of an experiment, encoding the coupling knowledge of the system;

(3) a set of level 4 general SESs describing the decomposition of the system into parts and variants and a corresponding set of level 4 general SESs describing the decomposition of experiments into individual tasks and alternatives, encoding the decomposition knowledge of the system; and

(4) a level 5 rule–based decision support system, which provides for the expert knowledge necessary to generate simulation models and simulation experiments for any given purpose, encoding the taxonomic knowledge of the system.

KNOWLEDGE ABSTRACTION

The previously described methodology of automated model synthesis is an entirely *deductive* process. The complete knowledge of the entire model taxonomy is encoded in parts and pieces at various levels of the five–level hierarchy. However, at least during its initial phases, modeling is a predominantly *inductive* process in which *structural knowledge* is inferred from *behavioral knowledge*. The reason for this seeming discrepancy is a concern with execution speed. Induction is usually too slow for real–time application. However, induction has also a place in a real–time distributed intelligent control environment, as will be shown in this section.

An event–based intelligent control architecture is envisaged. Event–based control is a discrete eventistic form of control logic in which the controller expects to receive conforming sensory responses to its control commands within prespecified *time–windows*.

The following simple scenario illustrates the advocated concept of a decentralized intelligent control architecture. The task to be solved by the envisaged high–autonomy intelligent control system is to have the system boil two three–minute eggs in the shell. The commander of the experiment (the customer) has a fairly decent idea as to how long this task may take, certainly no less than five minutes, but certainly no longer than 15 minutes. This is called a *time–window* (Wang and Cellier, 1991; Zeigler, 1989). Each controller that issues a command expects an acknowledgment of successful completion of the commanded task within a prescribed time–window. Receiving the order, the cook commands an apprentice to "go and get a pot." The cook knows that this task can be completed in no less than 30 seconds, but it should not take longer than two minutes. Thus, the cook (first–level subordinate controller) operates on his or her own time–window. If the apprentice returns with a pot within the prescribed time–window, the cook proceeds to the next command, which is to fill the pot with water. Again, a time–window serves to ensure the correct execution of the command. If the apprentice does not return in time, the cook knows that something has gone awry and he or she knows even more: either the pot was not where it was supposed to be or the apprentice just "died." Consequently, while the customer (commander) does not receive his or her three–minute eggs within the prescribed time–window, it is not necessary for him or her to decide whether the stove was broken, or the kitchen was out of eggs, or the pot had a leak and the water was spilled. The actual cause of failure will have been detected at a lower level closer to its source.

In the advocated methodology, it is usually the hierarchically lowest controller (within the command resolution) that detects the anomaly and calls upon the reasoner to determine its cause, since each controller operates on a time–window of its own. This is true under the assumption that each lowest–level control loop is equipped with the necessary threshold–type sensors to report back the successful completion of the commanded task. If some of the required sensors are not present, either because they are too expensive or because the controlled quantity cannot be directly measured, a failure will be propagated to higher–level command and control loops and must be detected at such levels (Luh and Zeigler, 1991a, 1991b).

Each controller uses a so–called *operational model* to determine the next time–window. This operational model is not a model of the overall operation. It is a specific submodel synthesized to help the commander determine the correct time–window under the specific operational conditions under which it is currently functioning. Depending on the complexity of the process involved in performing the next command, the operational model can be a simple table–lookup function or can involve performing multiple simulation runs of a continuous–time model while varying one or several model parameters. Depending on the frequency of its use, the operational model is either precompiled or generated on the fly. The reasoner uses a so–called *diagnostic model*, i.e., another submodel generated to aid the reasoner in the diagnosis of the cause of the just–detected anomaly. Only information needed for the task at hand will be included in the model. Execution of the diagnostic model may be as simple as applying an inference engine to a number of measurement data or may involve debugging the system by executing a potentially long list of test commands. Since fault diagnosis is always an exception, most diagnostic models will be generated on the fly. However, diagnostic models are also used by watchdog monitors in scheduled maintenance activities. Such diagnostic models are usually precompiled.

The event–based intelligent controller operates in a mode similar to a sampled–data controller. Figure 5 compares the two control paradigms to each other.

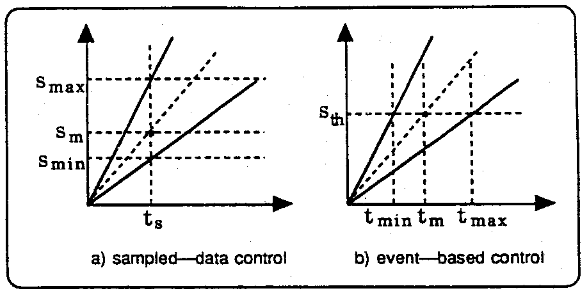

Figure 5. Comparison of sampled–data and event–based controller.

The sampled–data controller slices time into (usually equidistant) pieces and checks the status of the control system once during each sampling interval. The event–based controller schedules events to happen by discretizing the state variables themselves. For example, the event–based stove controller monitors the water temperature in the pot. In a commanded action, the stove may be requested to increase the water temperature to 100 °C. It would be possible to create a time–window requesting an acknowledgment of successful completion. However, it may be desirable to split the temperature interval into smaller portions of 10 °C and request one acknowledgment per successful completion of a subinterval.

The event–based operational model of the stove operates on knowledge that is automatically being abstracted from an equivalent continuous–time operational model of the stove. Whenever one of the local controllers needs a new time–window that cannot be found in its data base, it sends a request to the time–window generator. The *time–window generator* checks in its own data base whether it possesses the necessary submodel to extract the requested time–window. If this is not the case, the time–window generator commands the five–level hierarchy to generate the required submodel from the world model. It then performs a simulation experiment on that submodel to extract the requested time–window. For this purpose, the model is simulated many times over a relatively short time span while varying the model parameters within their assigned tolerance ranges. This must happen in real time, i.e., while the control of the process is ongoing. A good strategy for varying the model parameters is the sensitivity–in–the–large analysis technique, described in (Cellier, 1986).

The time–window generator extracts behavioral knowledge from the continuous–time operational model, and condenses this knowledge into the more–highly aggregated form of a time–window. This time–window is then incorporated in an equivalent discrete–event operational model that forms a part of the event–based intelligent controller.

Simulation software, whether continuous (e.g., ACSL) or discrete–event (e.g., GPSS), produces behavioral information from structural knowledge, i.e., uses a given model (encoding the structure of a real system) to generate trajectory behavior.

However, by itself, trajectory behavior is not sufficient to support decision making. A comprehensive approach must address the following questions:

(1) How are models generated? Simulation software assumes the presence of a precoded model. Where does this model come from? How and on the basis of what information is it constructed?

(2) How is knowledge abstracted from behavioral information? How can such knowledge be usefully employed in decision making?

Viewed in this light, simulation is one of three activities: modeling, simulation, and knowledge abstraction, which are all needed for automated intelligent decision making. A front–end module generates the model to be used by the simulation software. A back–end module uses the simulation results for knowledge abstraction.

Front–end and back–end modules are similar in form. The front–end creates models by extracting and abstracting knowledge from real–world experiments. The back–end extracts and abstracts knowledge from simulation experiments, which, if the model is valid, are replicates of potential real–world experiments. Thus, modeling and simulation are actually two siblings that can be used iteratively to abstract knowledge from systems. In an automated intelligent decision making system, models cannot be static entities that are created off–line by a human modeler once and for all ahead of their use. Instead, the processes of modeling and simulation are two essential components in an automated knowledge–abstraction scheme. Models should be retrieved from a model base or generated on the fly for immediate use in a simulation experiment (Wang and Cellier, 1991). Depending on the problem, the simulation results may, in turn, give rise to the generation of another model, and so forth. The end–product of a recursive application of modeling and simulation software is the abstracted knowledge that can be successfully used in decision making.

MODEL–BASED INTELLIGENT CONTROL OF AN ASTEROIDAL OXYGEN PRODUCTION PLANT

Utilization of lunar, planetary, and asteroidal resources has emerged as a challenging goal of space exploration (Cutler, 1988). Space mining and ore processing must necessarily involve a high degree of automation and system autonomy (Gothard and King, 1989; Suitor *et al.*, 1990). Although

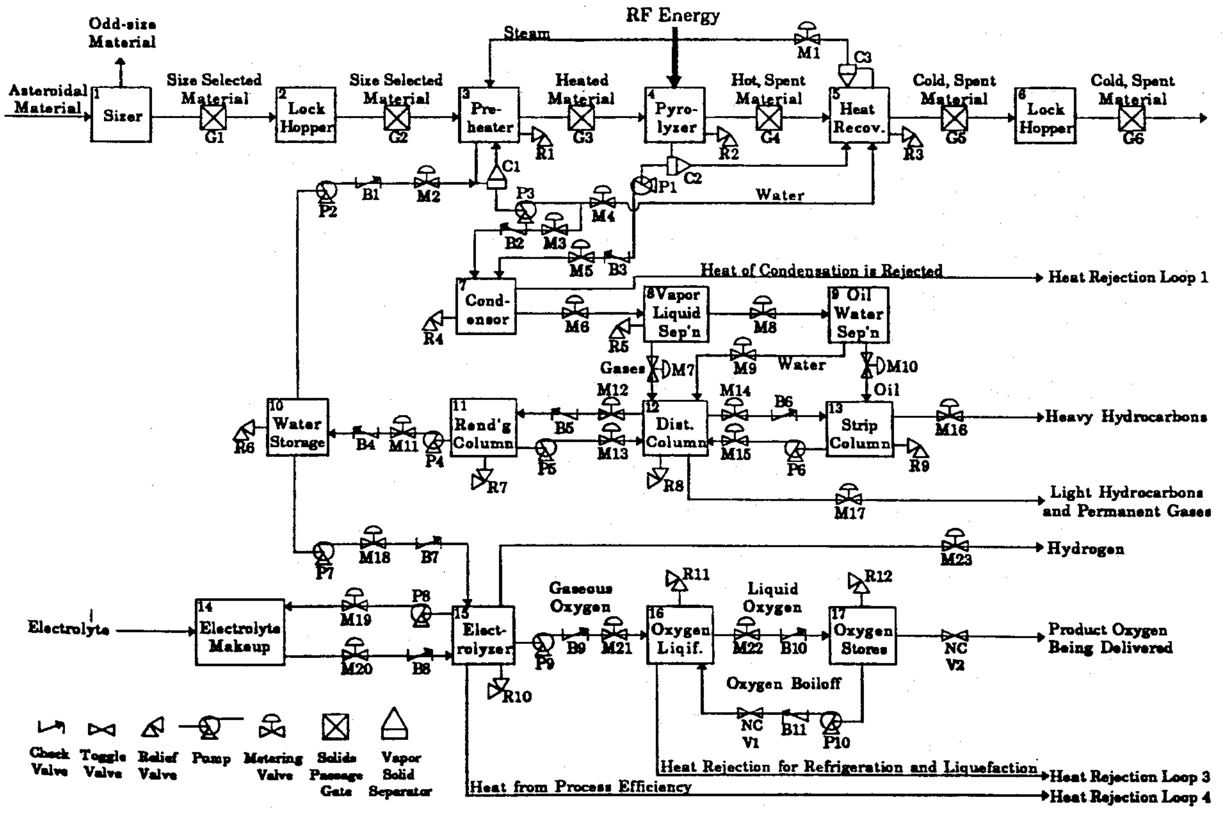

Figure 6. Oxygen production from chondritic asteroids.

the environments present different challenges, both terrestrial and extraterrestrial automation of mining systems involve the same basic processes (Schnakenberg, 1988) and advanced automation technologies developed for space are bound to have application on Earth as well.

Many near-Earth asteroids are likely to be rich in carbonaceous chondritic material. Water vapor can be extracted from such material in a pyrolysis stage. The water content of such chondritic asteroids is believed to be between 5% and 10%. Figure 6 shows a chemical process that could be used for oxygen production from chondritic material (Cutler, 1988, 1989, 1990).

The sizer consists of a crusher that decomposes the rock mechanically and a sieve that lets sufficiently small material pass and reroutes larger pieces to the crusher for further decomposition. The material is then forwarded to an inlet lock hopper, which accepts the presized material under vacuum and forwards it to the preheater under pressure. The preheater brings the entire charge to a fixed temperature slightly below that at which pyrolysis begins to occur. The preheater is also responsible for controlling the gas pressure, which may rise due to beginning evaporation of the charge. The pyrolyzer devolatilizes the preheated feed by the addition of microwave heat. Rapid pressure control is important in order to keep the gas pressure inside the pyrolyzer within acceptable bounds at all times. The heat recovery stage is a heat exchanger that recovers a high percentage of the heat of the spent feed after pyrolysis. The temperature and pressure must be controlled carefully in order to recover as much heat as possible without condensing water vapor, which would be thrown out with the spent feed and would thereby be lost. The outlet lock hopper accepts the spent feed and discharges it to the vacuum with little loss of the pressurized gas inside the system. This completes the description of the top row of Fig.6.

The second row contains a condenser, which precools the pyrolyzate vapor. A vapor liquid separator sends the remaining gases to the third row for further processing and lets the liquid proceed to an oil-water separator. Both products of the oil-water separator are processed further in the third row. This completes the description of the second row.

The third row contains three columns. The distillation column accepts the remaining gases and the impure water from the second row. It separates the material further. The heavy material ends up at the bottom, while the lightest material is at the top. The bottom will contain hydrocarbons, which are fed to the stripping column for further processing. The top will contain gases that can either be disposed of or recycled for production of hydrogen. The medium portion of the distillation column contains prepurified water, which is passed on to the rendering column for further purification. The stripping column is used to dewater the heavy hydrocarbons. The rendering column removes the remaining heavy organic material from the prepurified water. Its top product is pure water, which is passed on to the water storage area to await further processing. This completes the description of the third row.

The fourth row mixes the pure water with an electrolyte, such as potash, KOH, for processing by the electrolyzer. The electrolyte will invariably get contaminated by impurities in the water and must therefore be periodically recycled to an electrolyte makeup chamber where the quality of the electrolyte is controlled and improved when needed. The electrolyzer produces hydrogen and oxygen. In order to prevent losses of the oxygen product due to leakage, an oxygen liquefier is used, which will convert the oxygen to a form that can more easily be stored in the oxygen storage area. Oxygen that boils off is captured and rerouted to the oxygen liquefier. The heat that is produced in both the electrolysis and the oxygen refrigeration stage is also rerouted to the top column where it can be reused. This completes the description of the fourth and final row.

Many things can go wrong in this system. The sizer can get clogged due to all sorts of problems, e.g., with electrostatic charges. Its mechanical parts can be worn. The lock hoppers can leak reducing the pressure in the preheater and pyrolyzer. Every single component can be faulty. Moreover, the plant contains several local control loops for controlling the temperature, pressure, electrolyte composition, etc. Any one of these control loops can operate outside its foreseen range. Such faults and anomalous operations provide the basis for the investigation of the advocated fault-tolerant intelligent control system, as shown in Fig. 7.

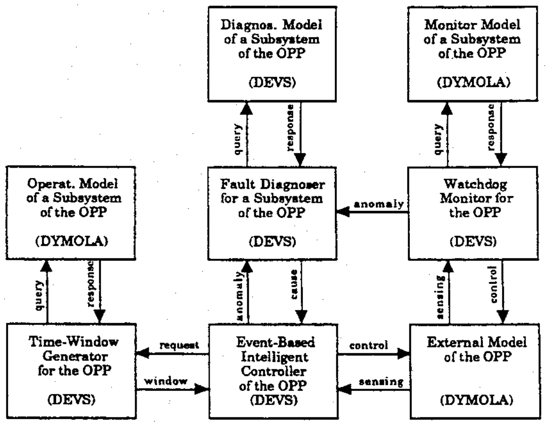
Diagnos. Model of a Subsystem of the OPP (DEVS)
Monitor Model of a Subsystem of the OPP (DYMOLA)
query
response
query
response
Operat. Model of a Subsystem of the OPP (DYMOLA)
Fault Diagnoser for a Subsystem of the OPP (DEVS)
anomaly
Watchdog Monitor for the OPP (DEVS)
query
response
anomaly
cause
sensing
control
Time-Window Generator for the OPP (DEVS)
request
window
Event-Based Intelligent Controller of the OPP (DEVS)
control
sensing
External Model of the OPP (DYMOLA)

Figure 7. Intelligent fault-tolerant control of oxygen production plant (OPP).

The *external model* of the oxygen production plant (OPP) represents the actual process. The testbed consists of a simulation only. Therefore, the plant itself must be represented by an adequately detailed model of the plant. This model is a continuous-time model. Its atomic components are coded in DYMOLA (Elmqvist, 1978). The coupling models are generated by DEVS-Scheme. The actual simulation is performed in either ACSL (MGA, 1986) or DESIRE (Korn, 1989), two of the continuous simulation languages for which DYMOLA can generate code.

The *event-based intelligent controller* of the OPP is coded in DEVS-Scheme. It performs all sensing and control actions necessary to monitor and command the OPP. The event-based controller is itself a hierarchical system consisting of individual local controllers for particular subsystems and more-aggregated supervisory coordinators responsible for the overall system performance. Each local controller operates on its own time-windows. Some of these time-windows are precompiled and stored in a data base, while others depend on the current operating conditions and need to be generated on the fly. Some of the time-windows result from transient responses to commands to be executed. Others are auxiliary time-windows caused by the monitoring of continuous processes. Some commands result from discrete actions, others are necessitated by monitoring needs. For example, the electrolyte must be sampled periodically in order to test for contamination. This can be accomplished by repetitive execution of an electrolyte-testing command.

Whenever one of the local controllers needs a new time-window that cannot be found in its data base, it sends a request to the time–window generator. The *time–window generator* checks in its own data base whether it possesses the necessary submodel to extract the requested time–window. If this is not the case, the time–window generator commands the five–level hierarchy to generate the required submodel at the 2nd hierarchy level from the world model coded at the 4th hierarchy level. It then performs a simulation experiment on that submodel to extract the requested time–window.

The *operational models* of the subsystems of the OPP are DYMOLA–coded models, similar to those used in the external model. However, the "control" experiment performed on the operational model is very different from the control experiment performed on the external model. The experiment performed on an operational model simulates that model several times with varying parameters over a fairly short time period to determine the next time–window. This must happen in "real time," i.e., while the external model simulates the overall process over time. For this reason, the corresponding signal paths in Fig. 7 have been named "query" and "response" rather than "control" and "sensing."

Each local controller contains a fault–detection agent, i.e., if the acknowledgment of completion arrives too early, or too late, or not at all, the local controller reports this fact as an anomaly to the fault diagnoser. The corresponding local *fault diagnoser* checks by means of shallow reasoning whether a unique cause for the anomaly can be determined. If this is not the case, the anomaly is propagated to the next higher supervisory level, and deep reasoning by means of symbolic simulation (Chi and Zeigler, 1991) is used to identify the cause of the anomaly. The supervisory diagnoser searches its data base for an appropriate diagnostic model to determine the cause of the reported anomaly. If such a model is not presently available, it commands the five–level hierarchy to generate the necessary diagnostic model. Once the required diagnostic model has been generated, the fault diagnoser performs a symbolic simulation experiment, which runs the diagnostic model through a number of hypothesized scenarios (resulting from shallow reasoning) to determine which of these scenarios is able to replicate the observed anomaly.

Contrary to the external and the operational models, the *diagnostic models* are discrete–event models coded in DEVS–Scheme. They contain all sorts of possible failures that can be either switched on or off depending on the experiment performed on these models.

The *watchdog monitors* are separate parallel units that monitor the external model for anomalies. Contrary to the fault detectors inside the control units, the watchdog monitors can be turned on or off on the fly without immediate effect on the overall control performance. Watchdog monitors emulate human plant supervisors in a control room whose purpose it is to discover potential problems in a control system. By turning some or all of the watchdog monitors on, the fault–tolerance of the control system is increased, since the watchdog monitors may be able to detect anomalies that cannot be discovered by any of the local control units. In particular, the local control units can only detect anomalies resulting from incorrect execution of a commanded action. However, faults can occur that are not related to a commanded action, but that are related to the break–down of equipment during steady–state operation. Such faults are detectable by the watchdog monitors. The "control" signal back from the watchdog monitor to the external model symbolizes scheduled maintenance activities.

The watchdog monitors operate on their own models by comparing measured plant behavior with expected model behavior to discover anomalies. They are also responsible for maintaining a logbook. The *logbook* contains records of previously observed anomalies (symptoms) together with the identified causes of these anomalies (failures) and the performed actions (repair activities) to recover from these failures. The logbook is useful for the human plant operator, but it also supports the fault diagnosers since the search can be executed in the sequence of most frequent occurrence of the same symptom in the logbook. This reduces the time needed by the diagnoser to identify the cause of a once–detected anomaly, which is important under real–time conditions.

CONCLUSION

In this paper, it was shown how object–oriented, i.e., distributed modeling can help to organize the knowledge about a system to be described. A methodology was presented that enables to encode separately and in an organized manner functional, coupling, decomposition, and taxonomic knowledge about a system. This methodology lends itself to the implementation of automated procedures for deductive model synthesis. Tools for inductive model synthesis were also introduced. The paper culminated in the description of a decentralized intelligent fault–tolerant control system of a fairly complex chemo–physical material handling and processing plant.

REFERENCES

Booch, G. (1991). *Object–Oriented Design with Applications*, Benjamin/Cummings, Redwood City, CA.

Cellier, F.E. (1986). "Enhanced Run–Time Experiments for Continuous System Simulation Languages," in: *Proceedings, SCS MultiConference on Languages for Continuous System Simulation*, San Diego, CA, pp. 78–83.

Cellier, F.E. (1990). "Hierarchical Nonlinear Bond Graphs — A Unified Methodology for Modeling Complex Physical Systems," in: *Proceedings, European Simulation MultiConference*, Nürnberg, F.R.G., pp. 1–13.

Cellier, F.E. (1991). *Continuous System Modeling*, Springer–Verlag, New York.

Cellier, F.E., Q. Wang, and B.P. Zeigler (1990). "A Five Level Hierarchy for the Management of Simulation Models," in: *Proceedings, Winter Simulation Conference*, New Orleans, LA, pp. 55–64.

Chi, S.D., and B.P. Zeigler (1991). "Symbolic Discrete Event System Specification," in: *Proceedings, AI, Simulation and Planning for High Autonomy*, Cocoa Beach, FL.

Cutler, A.H. (1988). "Space Manufacturing," in: *Encyclopedia of Physical Science and Technology*, pp. 129–134.

Cutler, A.H. (1989). "Economic Exploitation of Near–Earth Carbonaceous Asteroids," in: *Automation of Extraterrestrial Systems for Oxygen Production*, Space Engineering Research Center, University of Arizona, Tucson, AZ, pp. II–10 to II–15.

Cutler, A.H. (1990). "Process Design and Automation Requirements," in: *Proceedings, Space'90 Conference*, Albuquerque, NM, pp. 23–26.

Davis, R., F.E. Cellier, C. Grim, and C. Shaw (1990). "Space Station Freedom Program — User Interface Language Specification," Version 2.0, NASA Standard Document: USE 1001.

Elmqvist, H. (1978). *A Structured Model Language for Large Continuous Systems*, Ph.D. dissertation, Report CODEN: LUTFD2/(TFRT–1015), Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Goldberg, A., and D. Robson (1983). *Smalltalk–80: The Language and Its Implementation*, Addison–Wesley, Reading, MA.

Gothard, B., and B. King (1989). "Automation of Materials Processing," in: *Automation of Extraterrestrial Systems for Oxygen Production*, Space Engineering Research Center, University of Arizona, Tucson, AZ, pp. IV–21 to IV–43.

Hu, J.F., Y.M. Huang, and J.W. Rozenblit (1989). "FRASES — A Knowledge Representation Scheme for Engineering Design," in: *Proceedings, SCS Eastern Simulation MultiConference*, Tampa, FL, pp. 141–146.

Kim, T.G. (1988). *A Knowledge-Based Environment for Hierarchical Modeling and Simulation*, Ph.D. dissertation, Dept. of Electrical & Computer Engr., University of Arizona, Tucson, AZ.

Korn, G.A. (1989). *Interactive Dynamic–System Simulation*, McGraw–Hill, New York.

Luh, C.J., and B.P. Zeigler (1991a). "Abstraction Morphisms for Multifacetted Methodology: Application to Model–Based Autonomous Systems," submitted to *IEEE, Trans. Systems, Man, and Cybernetics*.

Luh, C.J., and B.P. Zeigler (1991b). "Abstraction Morphisms for Task Planning and Execution," in: *Proceedings, AI, Simulation and Planning for High Autonomy*, Cocoa Beach, FL.

MGA (1986). *ACSL Reference Manual*, Mitchell & Gauthier Associates, Concord, MA.

Rozenblit, J.W., J.F. Hu, and Y.M. Huang (1989). "An Integrated Entity–Based Knowledge Representation Scheme for System Design," in: *Proceedings, NSF Design Research Conference*, Amherst, MA, pp. 393–408.

Schnakenberg, G.H. (1988). "U.S. Bureau of Mines Coal Automation Research," in: *Proceedings, 3rd Canadian Symposium on Mining Automation*, Montreal, Canada.

Shafer, D. (1988). *Hypertalk Programming*, Hayden Publishing, Carmel, IN.

Suitor, J.W., W.J. Marner, F.E. Cellier, and L.C. Schooley (1990). "Automation and Control of Off–Planet Oxygen Production Processes," in: *Proceedings, Space'90, Engineering, Construction, and Operation in Space*, Vol. 1, ASCE, New York, pp. 226–235.

Wang, Q., and F.E. Cellier (1991). "Time–Windows: An Approach to Automated Abstraction of Continuous–Time Models into Discrete–Event Models," *International Journal of General Systems*, special issue on modeling and simulation of high–autonomy systems, to appear.

Zeigler, B.P. (1987). "Hierarchical, Modular Discrete–Event Modeling in an Object–Oriented Environment," *Simulation*, **49**(5), pp. 219–230.

Zeigler, B.P. (1989). "The DEVS Formalism: Event–Based Control for Intelligent Systems," *Proceedings of IEEE*, **77**(1), pp. 27–80.

Zeigler, B.P. (1990). *Object–Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, Boston, MA.

Zhang, G., and B.P. Zeigler (1989). "The System Entity Structure: Knowledge Representation for Simulation Modeling and Design," in: *Artificial Intelligence, Simulation and Modeling* (L.E. Widman, K.A. Loparo, and N.R. Nielsen, eds.), John Wiley & Sons, New York, pp. 47–73.