Proceedings of the 1991 Winter Simulation Conference
Barry L. Nelson, W. David Kelton, Gordon M. Clark (eds.)

INCORPORATING SIMULATION INTO A DESIGN ENVIRONMENT

Christopher Landauer

Computer Science Research Department
The Aerospace Corporation
Los Angeles, California, 90009-2957

ABSTRACT

This paper describes a methodology that provides a simulation capability for certain kinds of software environments. The required simulation functions are partitioned into a number of well-defined functional roles that represent basic units from which discrete event simulation programs can be constructed.

We use a new approach to constructing modelling environments. To insulate the users from many of the software integration details, the environment uses explicit knowledge of its own structure to support the user in selecting and adapting the system components. The knowledge is in the form of "wrappings", which are expert interfaces to the programs, tools, and other resources in the environment. This approach is a simple and powerful mechanism for making many different kinds of resources available to work together in an integrated way.

This paper defines the functional roles in an existing set of discrete event simulation support functions, and describes wrappings for them. It also shows how ordinary differential equation solvers fit into the same framework.

1 INTRODUCTION AND BACKGROUND

This paper will show how to wrap discrete event simulation control mechanisms to make them explicitly available in a modelling environment. The wrappings are part of a methodology for building flexible environments by encapsulating both programs and data with the explicit knowledge they embody, and with knowledge of their various styles of use (see [Bellman,Gillam 90], [Landauer 90], [Walter,Bellman 90]). These wrappings provide standard interfaces to software resources, and provide knowledge about the resource, so that other tools in the environment can interact appropriately with the wrapped resource, either to provide it with information or to use its information effectively.

The original context of this problem is the Vehicles system (see [Bellman,Gillam 88], [Gillam 89]), a large software environment supporting conceptual design of spacecraft, but the approach applies more generally to any software environment containing a heterogeneous collection of software resources (programs, databases, computational tools, interfaces, simulation models).

The Aerospace Corporation's role in defining new space systems and analyzing new space missions emphasizes the need to perform concept exploration studies that use diverse models, including simulations, analytic equations, and other software programs. The key to effective modeling in this context is a software environment that supports flexible use of a wide variety of models to help pose and answer study questions. This breadth requirement leads to the incorporation of many heterogeneous methods and software resources, including analysis tools, external programs, datasets, and models. This flexibility requirement leads to the need for rapid integration of models and other resources.

In order to provide a rich environment in the early stages of complex system design, many of these resources must be allowed to interact with each other, as well as with the conceptual design being built or examined. It is clear also that access should be provided to other analysis programs, including those developed externally.

The power of a complex software environment lies in its collection of models and modelling paradigms (in this context, the programs and databases are also models), and it is clear that harnessing that power effectively is a difficult problem. An important principle that has emerged from the work on Vehicles is that the appropriate flexibility requires all of the models to be made explicit, and that they be explicitly described in ways that can be processed by the environment. These descriptions are called wrappings (see

[Landauer 90]); they are fundamental to our methods of allowing the environment to assist the user in selecting appropriate tools, models, or other resources, adapting them to the study context, and interpreting their results.

The "wrapping" methodology builds flexible environments by encapsulating both programs and data with the explicit knowledge they embody, and with knowledge of their various styles of use. These wrappings provide standard interfaces to software resources, and provide knowledge about the resource, so that other tools in the environment can interact appropriately with the wrapped resource, either to provide it with information or to use its information effectively. An overview of the wrapping methodology is presented elsewhere in this conference (see [Bellman 91]).

One of the important modelling paradigms in conceptual design is discrete event simulation, and this paper will describe our methods for making simulation functions available in the design environment.

2 DESCRIPTION OF WRAPPING

A wrapping is an expert interface that describes a software resource (see [Landauer 90], [Bellman 91]). The wrapping contains a large amount of self-description, including knowledge of what the resource does, when it is appropriate to use it, how it should be used, and what information and performance can be expected from it. We are developing a general approach to wrapping programs and other software resources, including the notations necessary to describe wrapped resources to other components of a system, methods for using those wrappings, and the relationships between problems to be solved and resources that may be relevant to that problem.

The wrapping methodology is a new way to produce open integrated software environments. While the methodology has not been formulated completely, certain principles are clear. Everything gets explicit descriptions, computational analysis tools, simulation programs, databases, analytic and structural models, user interfaces, and other external programs and data, including the programs that interpret the wrappings of other software. Any part of a complex software environment is considered to be a software resource.

The descriptions are tailored to expected uses, so that the same software may have many different descriptions. A program may have many partial descriptions; complete descriptions are not needed (they are often much too complicated anyway). Descriptions for human users of the software are different from descriptions for programs that use the software, since they ask different questions of a resource, and can be expected to know different things about it. Integration occurs at many levels, from transferring bytes between resources on (possibly) different machines at the low level to exchanging information at the high level. It is more than making it so that programs can talk together; it also involves determining when it is appropriate for them to do so. These and other principles are being sharpened and refined into a description of the methodology.

The methods we have devised apply quite generally to any software and data in a heterogeneous environment. Furthermore, this wide an outlook is necessary to make full use of the wrappings, since the integration process involves all of the resources, and since the programs that process wrappings are as fundamental to the integration process as the tools and methods being integrated.

3 SIMULATION MODELS

There are many implementations of individual simulations in modelling environments, and even many implementations of simulation languages or other construction mechanisms, but all of them make certain assumptions about the kind of scheduling and event management used; some of the better ones allow a few choices of event management functions, but they are typically only a very few. We want to make explicit all choices of models, including those that support others, such as the event management functions in simulation models, ordinary differential equation solvers, and other numerical analysis algorithms used, because they are not (always) neutral; their assumptions are often important.

For example, the scheduling mechanisms in most simulation languages include the assumption that if two events are scheduled to occur at the same simulation time, then the one that was scheduled first in real time is the one that will be invoked first. This assumption provides a zero-delay communication channel which can (and often is) be abused to give unrealistic effects in simulation models (for example, the first event sets a value that the second event can read). Some languages circumvent that channel by separating the events that are scheduled for the same simulation time and selecting from them in a special way (randomly, in many cases). Our modeling experience shows that making the assumption an explicit part of the model helps prevent this kind of abuse.

4 EXAMPLE IMPLEMENTATION

The simulation mechanisms we use are based on a common set of support functions (see [Landauer 85]) that handle the time and event management functions. The support functions are written in *C*, and run on many *UNIX* systems. These functions fall into two classes: those that occur before the simulation starts, and those that occur while it is running.

The simulation support functions assume that each indivisible event is defined by a separate *C* function call, so that, for example, interruptible events are modeled by several event functions.

Simulation time in the simulation program is managed by a main loop that checks for interrupts (these simulation programs are interactive, and allow user- or program-generated interrupts at any time), determines the next event to occur (via the event management functions described below), updates the current simulation time, and calls the appropriate event function to act out the event.

The simulation initialization functions include a function that defines an event to the system (by defining its event function), and a function that defines a monitor to the system (each interrupt is distributed to an appropriate user interrupt monitor).

The event management functions include selecting the next event to occur, and scheduling a new event to occur some simulation time in the future.

The functional roles that normally occur before the simulation run are:

- define an event function,
- define a user interrupt monitor,
- add event at a specified simulation time in the future.

The last of these can occur before or during the simulation; here it defines the initial set of events. The only other process that precedes the simulation run is initializing the simulation time. We will not treat the interrupt monitors further in this paper.

The functional roles that normally occur during the simulation run are:

- add event at a specified simulation time in the future,
- check for outstanding interrupts,
- select event to occur next,
- update simulation time,
- invoke event function.

The first of these can occur before or during the simulation; the rest comprise the main loop. We will not treat the interrupt processes further in this paper.

There is nothing in the system that precludes defining new events during the simulation run; that capability and others (the interrupt processing and user interface, for example) are beyond the scope of this paper.

5 SIMULATION CONTROL PROCESS

This section will describe one typical decomposition of the simulation control process into a set of fundamental building blocks (or functional roles), show how different implementations of the building blocks (different functions) can be put together in different ways, and provide example wrappings of some of the roles and functions. The wrappings contain enough information to determine when the composition of the building blocks makes sense, when certain capabilities are required in the functions, which sets of capabilities are compatible and incompatible, and how the functions pass data and control to each other.

5.1 Functional Roles

The functional roles we will wrap are as follows, listed with their input and output parameters:

- defnevnt - define an event reference
  - inputs - event function, event name
  - outputs - event reference
- initevch - initialize current event schedule and simulation time
  - inputs - none
  - outputs - event set reference
- schedule - add event at a specified simulation time in the future
  - inputs - event reference or event name, event set reference
  - outputs - success flag
- nextevnt - select event to occur next
  - inputs - event set reference
  - outputs - event reference
- updtsimt - update simulation time
  - inputs - event reference, event set reference
  - outputs - simulation time

- invkevfn - invoke event function
  - inputs - event reference
  - outputs - success flag

An event reference is an internal data structure that contains the event name, the event function pointer, and whatever statistics about the event invocation that we choose to keep. An event set reference is an internal data structure that keeps track of the current set of scheduled events and the simulation time. Each run of a simulation program uses one instance of this structure. With this as an explicit structure, we can allow several simulation runs to be concurrently executed (there are still some event function implementation details to consider for separation of global variables and auxiliary function data, but these are a well-known part of writing re-entrant programs).

These functions and roles have the following information in their wrappings. These descriptions are in an informal version of the wrapping language, which is still under development.

5.2 Wrapped Process Descriptions

The first set of definitions describe our decomposition of the simulation processes into sequences of steps. In all cases, any quoted string is a comment that denotes a text description of the process, role, rule, or function, that can be displayed to the user.

The "repeat" control structures in many of the process descriptions have no explicit condition for termination. The assumption is that the associated process is not controlled by the steps listed; they form a pattern of activity that must conform to the process. The process is driven from the outside.

Also, whenever a condition or a function does not have an explicit definition, either the system or the user will be asked to supply the information. These wrappings are not a complete definition of the resources; they are a description that can be used in many ways.

```
PROCESS     simulation
            "define and run a simulation"
RULES       curevstf, futevstf
            "rules are prerequisites for
                using a process"
STEPS       simsetup; simcntrl;
END

PROCESS     simsetup
            "define events and initialize
                a simulation"
STEPS       repeat { defnevnt; }
            initevch;
            repeat { schedule; }
END

PROCESS     simcntrl
            "simulation main loop"
STEPS       while (nextevnt)
                {
                updtsimt;
                if (endtime) break;
                invkevfn;
                }
END

FUNCTION    endtime
            "has simulation time exceeded
                termination time?"
OUTPUT      Boolean
END
```

5.3 Wrapped Functional Roles

The next definitions describe the functional roles discussed above. There will be type declarations for event_function, event_name, event_reference, event_set_reference, null_event, and simulation_time (we use an abstract data type definition to define data structures; its details are not important here). We call these functional roles and not just functions because we want to allow different sets of functions to map into these roles. Indeed, for each functional role, there may also be a process description, since in some cases, the functional role must be decomposed into smaller roles before it can be mapped into a particular function.

```
ROLE        defnevnt
            "define an event reference"
INPUT       event_function, event_name
OUTPUT      event_reference
END

ROLE        initevch
            "initialize current event set
                and simulation time"
INPUT       none
OUTPUT      event_set_reference
END

ROLE        schedule
            "add event instance at a
                specified simulation time
                in the future"
INPUT       event_reference or event_name,
                event_set_reference
OUTPUT      Boolean
END
```

```
ROLE        nextevnt
            "select event to occur next"
INPUT       event_set_reference
OUTPUT      event_reference or null_event
END

ROLE        updtsimt
            "update simulation time"
INPUT       event_reference,
                event_set_reference
OUTPUT      simulation_time
END

ROLE        invkevfn
            "invoke event function"
INPUT       event_reference
OUTPUT      Boolean
END
```

The roles of defnevnt, updtsimt, and invkevfn will not be decomposed, since they correspond fairly directly to particular functions.

5.4 Event Management Processes

There are two event management choices to make: whether or not to use a separate current events set, and whether to use a heap or a list data structure for the future events set. Here, curevset and nocurevs are opposing values for the context flag curevstf, indicating, respectively, that there is or is not a separate current events set. The conditionals below are analogous to LISP conditionals.

```
RULE        curevstf
            "separate current events set
                or not"
COND        "expect many simultaneous
                events or not"
IMPL        curevset
COND        "insist on precluding zero-time
                information transfers"
IMPL        curevset
ELSE        nocurevs
END
```

Now we describe how the first context flag affects the functional roles for initevch, schedule, and nextevnt. Here, "PRECOND" marks a pre-condition that must be true before the process can be elaborated in the steps indicated. We also note that the schedule process is not conditional; it is always elaborated as addfutev (since our implementation always adds an event to the future event set).

```
PROCESS     initevch
PRECOND     nocurevs
STEPS       initfuts;
END

PROCESS     initevch
PRECOND     curevstf
STEPS       initfuts; initcurs;
END

PROCESS     schedule
STEPS       addfutev;
END

PROCESS     nextevnt
PRECOND     nocurevs
STEPS       futevnt;
END

PROCESS     nextevnt
PRECOND     curevstf
STEPS       if (not curevnt)
                {
                if (futevnt)
                        {
                        futevnt;
                        addcurev;
                        while (sametime)
                                {
                                futevnt;
                                addcurev;
                                }
                        }
                else fail;
                }
            curevnt;
END

FUNCTION    sametime
            "is current event time same as
                first future event time?"
OUTPUT      Boolean
END
```

The function initcurs initializes the current events set. The function curevnt checks for a non-empty current events set and takes one randomly as the next event to use. The function addcurev adds the current event to the current events set. The implementation of the current events set is entirely hidden.

Finally, we describe the other context flags, heap_set and list_set, which are used to select functions for the roles of initfuts, futevnt, and addfutev. The role of initfuts is to initialize the future events set. The role of futevnt is to check for a non-empty

future events set and take the earliest one as the next event to use (there is no assumption about what happens with ties). The role of addfutev is to add one event to the future events set, at the specified time. The implementation of the future events set is entirely hidden within these roles. The mapping from these roles and the context flags to specific functions that implement the roles with a linear list or a heap is easily made.

```
RULE      futevstf
          "future events set structure"
COND      "event are scheduled for both
                short and long intervals"
IMPL      heap_set
COND      "event are scheduled mostly
                for short intervals"
IMPL      list_set
ELSE      list_set
END
```

There is an overall implementation choice to make about whether to use these simulation support functions, or another set of functions such as SIMPACK (from Paul Fishwick at the University of Florida), or nest 2.5 (from Alexander Dupuy and Jed Schwartz at Columbia University). They have somewhat different assumptions and decompositions, but the overall structure can be described with the above mechanisms. As a user gains experience with these functions, more conditions on the appropriate choice for a problem can be recorded, so that eventually the system can assist a user in making the choice.

6 DIFFERENTIAL EQUATION SOLVERS

A very common special case of discrete event simulation uses systems of ordinary differential equations (ODEs) to model dynamical systems. Good ODE solvers like the usual Runge-Kutta methods or the Bulirsch-Stoer methods (which interpolates with rational functions instead of polynomials; see [Press et al. 86]) can be incorporated into a simulation model to propagate a differential system from one time point to another.

Both of these fourth-order solvers use a variable step size, and monitor their own accuracy by the use of a fifth-order check equation. The time steps define discrete events, which can be interleaved with the usual kind of events in a time-stepped discrete event simulation program, and various alarm conditions on the step size or accuracy estimates can be used to schedule other events that will examine or repair the problem.

Both of these functions use explicit start and stop times, estimated first step size, minimum allowed step sizes, and required accuracy. They also expect a derivative evaluation function and an initial value to be supplied.

In this case, the function calls have the same format and the same parameters, so the only selection criteria are accuracy of estimation and time required. The Runge-Kutta algorithm tends to take many more steps, each of which is much simpler. The Bulirsch-Store method appears to be more accurate, but only in long time steps. If we need to have accurate intermediate values, for plotting or spatial interactions, the Runge-Kutta method appears to be superior. These considerations would be written explicitly into the wrappings of the two functions, so that the role of "propagate a differentiable dynamical system" could be attached to either function according to context.

7 CONCLUSION

This paper applies our general notion of wrapping to discrete event simulation models, and shows how dynamic models can be selected and integrated into a large modelling environment. This method provides a simulation capability in a design environment, instead of just incorporating a set of simulation programs.

REFERENCES

[Bellman 91]. Kirstie L. Bellman, "Flexible Software Environments and the Design of Complex Systems", *These Proceedings*

[Bellman,Gillam 88]. Kirstie L. Bellman and April Gillam, "A knowledge-based approach to the conceptual design of space systems", *Proceedings 1988 SCS Eastern Multi-Conference*, pp. 23-27, The Society for Computer Simulation (March 1988)

[Bellman,Gillam 90]. Kirstie L. Bellman and April Gillam, "Achieving Openness and Flexibility in Vehicles", pp. 255-260 in *Proceedings of the SCS Eastern MultiConference*, 23-26 April 1990, Nashville, Tennessee, Simulation Series, Volume 22(3), SCS (1990)

[Gillam 89]. April Gillam, "A knowledge-based approach to planning the design of space systems", *Proceedings 1989 SCS Eastern Multi-Conference*, The Society for Computer Simulation (1989)

[Landauer 85]. Christopher Landauer, "Network and Protocol Modeling Tools", Proceedings IEEE/NBS Computer Networking Symposium, Gaithersburg, Maryland, December 1984, p. 87-93 (December, 1984)

[Landauer 90]. Christopher Landauer, "Wrapping Mathematical Tools", pp. 261-266 in *Proceedings of the SCS Eastern MultiConference*, 23-26 April 1990, Nashville, Tennessee, Simulation Series, Volume 22(3), SCS (1990)

[Press, et al. 86]. William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vettering, Numerical Recipes: The Art of Scientific Computing, Cambridge University Press (1986)

[Walter,Bellman 90]. Donald O. Walter, Kirstie L. Bellman, "Some Issues in Model Integration", pp. 249-254 in *Proceedings of the SCS Eastern Multi-Conference*, 23-26 April 1990, Nashville, Tennessee, Simulation Series, Volume 22(3), SCS (1990)

AUTHOR BIOGRAPHY

CHRISTOPHER LANDAUER is a researcher in the Computer Science Research Department of The Aerospace Corporation. His research interests include formal specification and verification of communication protocols, analysis and modeling of distributed computing systems, pattern detection and tracking from noisy data, mathematical experiments related to these applications. He has published several papers in the fields of validation of expert systems, the use of mathematical tools within expert systems, and high-level simulation modeling and statistical information retrieval.