Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

CREATE!: AN OBJECT-ORIENTED IDE FOR DISCRETE EVENT SIMULATION

Michael Rüger
Thomas Behlau

Department of Computer Simulation and Graphics
Otto-von-Guericke University of Magdeburg
P.O. BOX 4120
D-39016 Magdeburg, GERMANY

ABSTRACT

Create! is a graphical integrated development environment (IDE) for discrete event simulation. It features a complete modelling and simulation environment for end users working in a dedicated environment as well as an integrated development environment enabling advanced users or simulator developers to create extended or even new simulation environments.

This paper presents major aspects of the Create!-environment and some tools built using it.

1 INTRODUCTION

Through recent years new graphical simulation tools have appeared on the market with remarkable success in the simulation community. Most of these environments come with a set of prepacked components aimed at a certain application field and are more or less extensible.

Solutions to the problem of extensibility range from simple programming language interfaces providing access to external C(++) or Pascal code to combined library and simulation language approaches for extending the set of simulation elements. Finding a both flexible and intuitive way of specifying the behavior of elements or processes is another challenge in this context.

Allowing a user to create new or modify existing elements or process definitions implies the existence of a language or process specification of some sort to describe the desired behavior. Both compiling/linking on the one hand and interpreting this code on the other have their pitfalls. While the first generally produces faster models, it introduces a potential danger of corrupting the system through undesired side effects or simply programming mistakes. Interpreting user code allows trapping errors at runtime, but is somewhat slower.

Another important issue is the range of supported platforms. When implementing a system as complex as a simulation environment one has to deal with all aspects like releasing versions and providing easy-to-use graphical user interfaces for all platforms. Maintaining an application simultaneously on different platforms is both expensive and error prone.

By choosing Smalltalk as the underlying development environment, most of the above problems could be solved quite easily. ObjectWorks™-Smalltalk is binary portable across all supported platforms, thus allowing us to use Create! on workstations as well as on PCs. The incremental compilation and dynamic binding provides a way to add code during runtime while providing the safety of an interpreted system.

In the following we will show how these capabilities are used to solve the problems mentioned above and together with the modelling and development framework form a powerful environment with low threshold and high ceiling.

2 THE CREATE! CLOUD

The word *cloud* came up during the early Create! development, when there was just this cloudy vision, but no idea how to realize it. Realizing it produced a two step approach to the development of simulators. The first step is the creation of

- a generic runtime environment for building models, performing simulation runs and evaluating the results and
- an integrated development environment containing all necessary tools to create new libraries of elements and objects.

Combining the libraries with the generic simulation environment in the second step results in a new simulator for the end user (Figure 1).

Given access to the development tools, advanced users can modify existing elements or complement the environment with new ones.

Building new or extending existing simulators is thus accomplished by the same means.

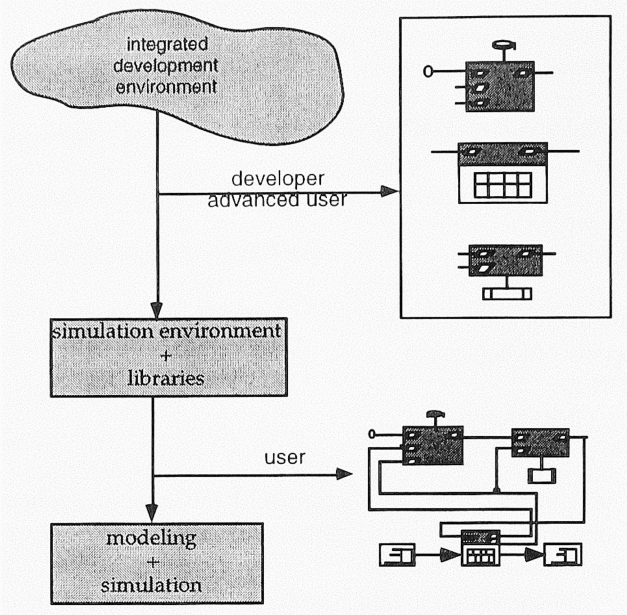

Figure 1: Create! IDE and Runtime Environment

3 MANAGEMENT FRAMEWORK

When building and running models the user has to deal with the management of data files, models and evaluation results. Organizing these manually into files and directories has two major drawbacks:

- dealing with platform specific filename conventions
- keeping track of versions.

Create! provides the concept of a simulation project to support the management of models, associated parameter sets and the results of simulation runs.

A simulation project is organized into studies, which in turn contain models and experiment series. Models come in two flavors: working versions and frozen ones. Frozen models may not be modified and form the basis for experiments, which therefore can be replicated at any time. Each of these project parts can be created, renamed, duplicated or deleted without the need to use the commands as provided by the underlying operating system. Figure 2 shows the project management window for the project "Distribution". The currently active study regarding Germany contains several models, part of which are frozen.

Projects provide yet another functionality. Each project forms a container for the installation of element and type definitions, libraries and icons. In this way, parts of an environment, which are a specific extension for a simulation project, are encapsulated within this project without cluttering the overall system. When entering a project, these components are dynamically loaded and hidden again when leaving the project.

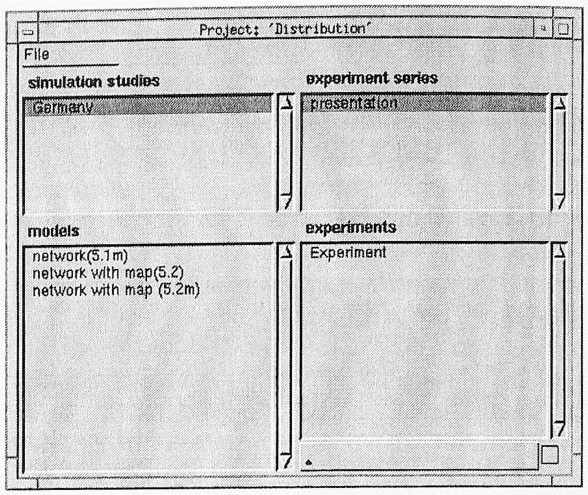

Figure 2: Project Management

4 MODELLING FRAMEWORK

There are two basic types of components in Create!:

- Elements. The "real world" items representing the active units of a model (productions cells, network nodes).
- Objects. The passive components (parts, packets) flowing through the system or forming the data structures used (processing definitions).

Models are built by placing elements in the model layout and connecting them (Figure 3). During the simulation, objects are passed along these connections and cause the execution of actions when entering the next element.

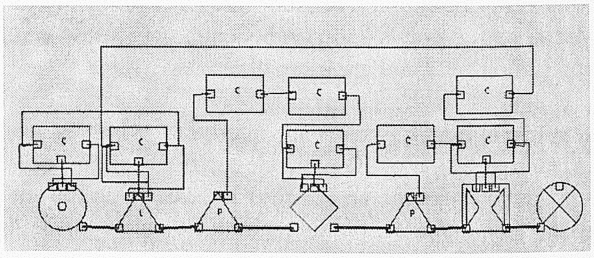

Figure 3: Create! Simulation Model

Elements consist of a runtime definition and an external representation. The external representation defines all properties needed for user interaction like icon, parameter dialog and graphical editing behavior (collision handling, auto-connect). The runtime specification is based on extended finite state machines with functions attached to the transitions (see "Specification Framework: ParSEC"). It also includes the internal data structures (parameters, variables). Both parts of the specification are kept externally on disk and are loaded into the simulation environment only when needed.

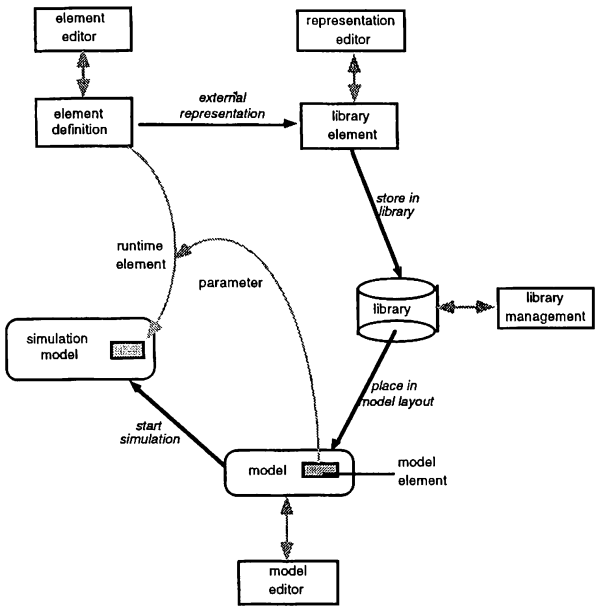

Figure 4: Element Lifecycle

A copy of the external representation of an element is stored in a library together with a reference to the runtime specification. When building models, only the element's representation is used. This allows us to keep different representations for the same runtime specification, e.g., presenting a set of preconfigured elements to the user by presetting some parameter values and removing the corresponding parameter entry fields in the dialog.

On simulation start elements are instantiated according to their runtime specification and the parameters in the model elements ("Element Lifecycle").

5 SPECIFICATION FRAMEWORK: PARSEC

Covering the problem of implementing a simulation kernel, a couple of different approaches were reviewed and analysed (Zeigler 1984, Paul 1993, Cota et al. 1994, Carrie 1988, SIMPRO 1985, Schruben 1983, Som and Sargent 1989).

As a result, the Create! simulation kernel has been based on extended finite automata. Each automaton is defined by a set of states, state transitions and functions attached to the transitions. The notion of finite automata is extended in so far as some transitions may be guarded, so that depending on the outcome of some boolean expression (condition), one of two alternate transitions will be executed.

This concept reduces the simulation kernel to a tiny mechanism for scheduling and distributing signals, and places hardly any restrictions on the kind of elements and models to be built.

Transition functions are defined (coded) using an object-oriented programming language which is syntactically close to C++ and conceptually close to Smalltalk-80™. It supports inheritance for data structures and simulation elements as well as function polymorphism depending on function name and parameters.

5.1 State Machines and Processes

The original SEC (State, Event, Condition) concept (Rüger et al. 1990) has been further extended to allow modelling of independent processes based on the SEC automaton definition (ParSEC: Parallel State, Event, Condition).

Therefore, an automaton definition is executed by a state machine which reacts to input signals and performs the appropriate transition depending on its current state. Several of these state machines can be forked for the same automaton definition, allowing several processes running independently, but with the same behavior.

Although the concept is named *parallel* SEC, processing of events takes place in a strictly sequential manner. Nevertheless, as each process of executing a state machine has its own execution space, the developer can think of these as executing in parallel without worrying about mutual exclusion on variable access etc.

An example for the deployment of parallel processes is a transport systems with several vehicles, all sharing the same behavior, but acting independently. The behavior of a vehicle would then be described by means of an automaton definition, but vehicles acting independently in the simulation model with a state machine forked for each one.

During the simulation each state machine is either in its current state or performing a state transition. Using *hold* or *wait* functions for introducing time delays would violate the concept of executing only one transition at a time. Therefore, signals can be postponed by scheduling them with a time delay. Signals, sent while executing a transition function, are queued and processed after completing the transition.

5.2 States and Signals

The basic elements of an automaton definition are states and signals.

The states of an automaton are a set of user defined symbols plus a predefined pseudo state *terminated*. A transition to this state terminates the currently active state machine and frees the corresponding slot in the process list.

Signals can be sent from within a transition or from a signal pin in one of the interface plugs (see below). Each signal carries exactly one object, which is provided as an argument to the invoked transition function.

5.3 Transition

A transition is defined by:

- initial state S
- next state S'
  (may be the same as S)
- signal s
- transition function f

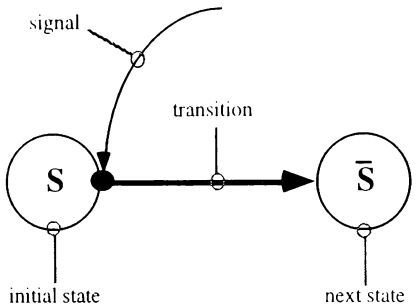

Figure 5: Transition

The transition function f is executed, if the state machine's current state is S and the signal sent is s. After executing f, the state machine's current state is S'.

5.4 Guarded Transition

A guarded transition is defined by:

- initial state S
- true state S'
  (may be the same as S or S'')
- false state S''
  (may be the same as S or S')
- signal s
- guard expression g
- true transition function f
- false transition function f'

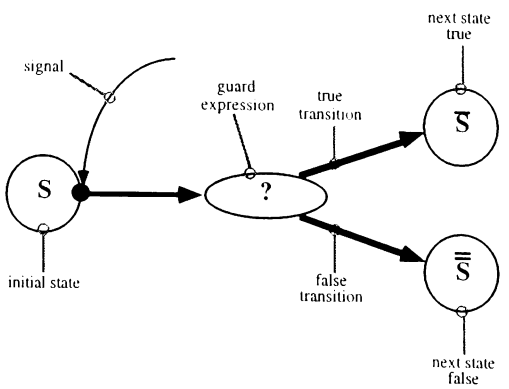

Figure 6: Guarded Transition

The transition function f is executed, if the state machine's current state is S, the signal sent is s and the evaluation of g returns *true*. After executing f, the state machine's current state is S'.

The transition function f' is executed, if the state machine's current state is S, the signal sent is s and the evaluation of g returns *false*. After executing f', the state machine's current state is S''.

5.5 Plugs and Sockets

Elements communicate with each other through plugs. A plug is defined by a set of pins and guards:

- Pin. A pin transmits a signal from one element to the other.
- Guard. A guard promotes a boolean value from one element to the other. It emits a signal in the connected element, when its value changes from *false* to *true*.

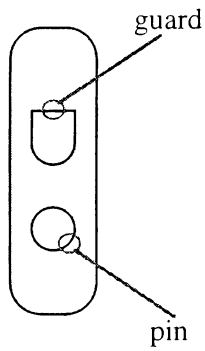

Figure 7: Simple Plug Definition

The very simple plug definition used in Figure 7 consists of:

- *Object* (output pin). It sends an object to the connected element.
- *Ready* (input guard). It signals if the connected element is ready to receive object. If the element becomes ready (sets the guard's value to *true*), a signal is sent.

When connecting two elements through a plug, the pins and guards are connected and promote the objects sent to a pin or guard in one element to the corresponding pin or guard in the connected element.

An *input* pin or guard receives a value from the connected plug and promotes it to the element's processes. An *output* pin or guard receives values from the element's processes and promotes them to the connected plug.

A plug definition defines a plug seen as an output interface. By reversing the direction of pins and guards a plug is turned into an input interface, which is then called *socket* (Figure 8).

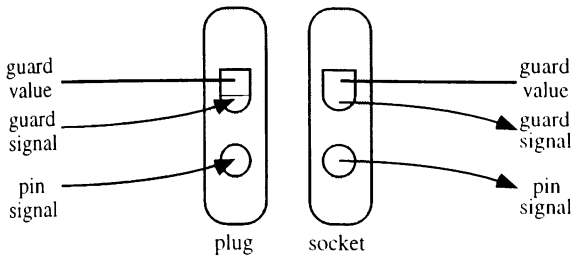

Figure 8: Simple Plug and Socket Pair

5.6 A Simple Example

The task of passing an object from one element to the next is illustrated in the following example.

After completing some task (leaving the state *busy* on signal *endBusy*), the element passes on an object to the connected element.

A guarded transition is used with:

S	*busy*
s	*endBusy*
g	check if the element is ready
f	output object
f'	do nothing
S'	*done*
S''	*wait*

Another transition is used for handling the delayed object passing:

S	*wait*
s	*ready*
f	output object
S'	*done*

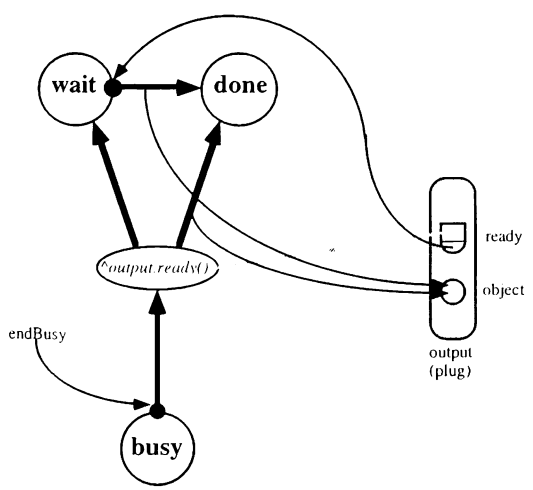

Figure 9: Guarded Transition Interacting with a Plug

6 EVALUATION AND ANIMATION

Evaluation and animation depend to a large extent on the application domain. Nevertheless, a set of basic evaluation and animation methods is desirable. Based on a generic framework, domain specific evaluations can be added when needed.

The solution chosen in Create! is to provide a method to generate a stream of events during the simulation and work on this stream online or offline (Rüger and Nyhuis 1992). The object-oriented nature of the underlying Smalltalk™ system makes it easy to transparently handle both internal and external streams or pipes.

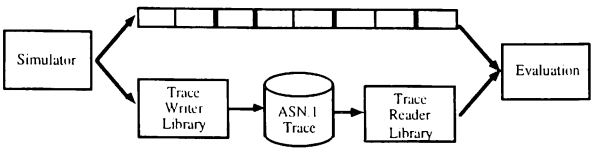

Figure 10: Trace Event Stream

The tracer interface allows element related trace events like *state*, *load* or *object count*, as well as object related events like creation, modification of attributes or deletion (Nyhuis 1994).

All events are piped through an evaluation network which consists of single data processors realizing tasks as counting, averaging or other statistical filter functions. Connected to the outputs of this network are visualization components like graphs or animation views. As evaluations only deal with events, they do not depend on the kind of elements generating these events, thus allowing us to provide a generic set of evaluation and animation methods.

7 EXISTING ENVIRONMENTS

Currently the following simulation environments have been built using the Create! IDE:

- Create!-Structure: simulation of logistic systems at structural level.
- Create!-LSG: simulation for strategic decision support.
- Create!-Batch: simulation for batch oriented production systems.
- Create!-Simple: simple single server environment for teaching purposes.
- Create!-LogiChain: modelling and static analyzing of processes (no simulation!).

All share the same runtime kernel with some minor simulator specific changes in the user interface. Figure 11 shows a snapshot of the model editor within the Create!-Structure environment with the list of the available library

elements on the left and the graphical working area on the right. The same editor is used in the other tools as well.

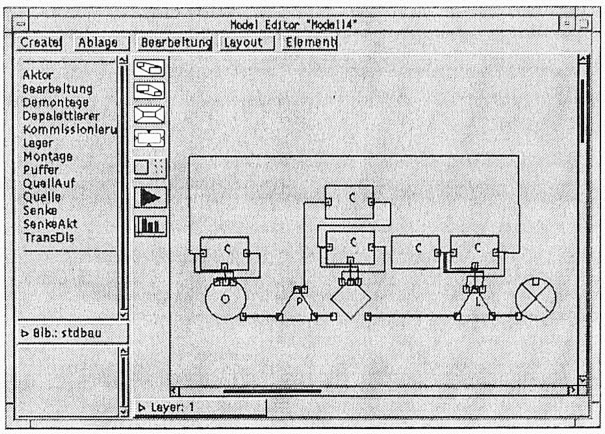

Figure 11: Create!-Structure Model Editor

A somewhat interesting exception is the Create!-LogiChain environment (Figure 12) which makes no use of the simulation kernel. It was built using the IDE's abilities to support graphical, element based modelling environments. Elements (activities) are placed on a regular grid. Controlled by parameters provided with the element definitions, elements of the process chain auto-connect to their next neighbors. After constructing the chain, a generic computation algorithm, which in turn calls functions defined within the elements, is run on the model and provides the results of the analysis.

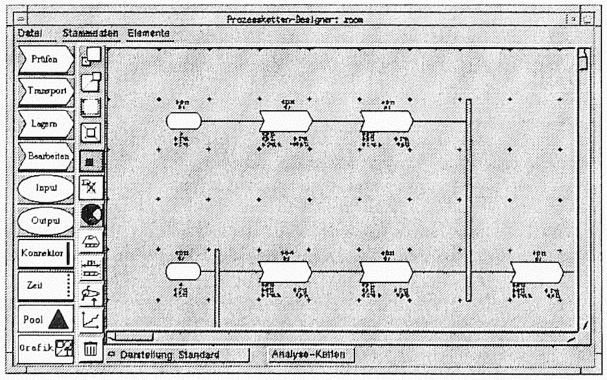

Figure 12: Create!-LogiChain Designer

REFERENCES

Carrie, A. 1988. *Simulation of Manufacturing Systems.* Chichester: John Wiley & Sons Ltd.

Cota, B.A., D.G.Fritz, and R.G. Sargent. 1994. Control Flow Graphs as a Representation Language. In *Proceedings of the 1994 Winter Simulation Conference*, 555 - 559, Lake Buena Vista, FL.

Nyhuis, A. 1994. Ein Trace in ISO-Normformat zur instrumentenunabhängigen Unterstützung objektbezogener Auswertungen. *Simulation und Integration,* ASIM Mitteilungen, Vol. 42, 26-34, Magdeburg.

Paul, R.J. 1993. Activity Circle Diagrams and the Three Phase Method. In *Proceedings of the 1993 Winter Simulation Conference*, 123-131, Los Angeles, CA.

Rüger, M., and A. Nyhuis. 1992. Monitoring und Animation in der Simulatorentwicklungsumgebung Create!. *Visualisierung und Präsentation von Modellen und Resultaten der Simulation*, ASIM Mitteilungen, Vol. 31, 132-140, Magdeburg.

Rüger, M., U. Hoppe, and H. Kirchner. 1990. Objektorientierte Modellierung von Bausteinen innerhalb der Simulatorentwicklungsumgebung Create!. *Fortschritte in der Simulationstechnik,* Vol. 1, 140-144, Vienna.

Schruben, L.W. 1983. Simulation Modelling with Event Graphs. *Communications of the ACM* 26: 957-963.

SIMPRO 1984. *GBS - Die graphische Basissprache für das SIMPRO-System.* Berlin: IMPRO GmbH.

Som, T.K., and R.G. Sargent. 1989. A Formal Development of Event Graph as an Aid of Structured and Efficient Simulation Programs. *ORSA Journal on Computing*, 1, 2, 107-125.

Zeigler, B.P. 1984. *Multifaceted Modelling and Discrete Event Simulation.* New York: Academic Press.

AUTHOR BIOGRAPHIES

MICHAEL RÜGER is a research staff member at the Department of Computer Simulation and Graphics at the University of Magdeburg, Germany. He received his Diploma in computer science from the University of Dortmund in 1988. After that he worked on simulation development and application at the Fraunhofer Institute for material flow and logistics in Dortmund and since then he has been working in the field of human computer interaction and visualization techniques at the University of Magdeburg.

THOMAS BEHLAU works as a graduate research assistant in the Department of Computer Simulation and Graphics at the University of Magdeburg, Germany. His areas of research are the modelling of manufacturing systems and modelling methodologies. He is a member of the GPSS-Users'-Group Europe.

World-Wide-Web

The Create! home page can be reached at
http://simsrv.cs.uni-magdeburg.de/~create