

# SPECIFICATION AND ANALYSIS OF CONCURRENT SYSTEMS IN A COMPLETE VISUAL ENVIRONMENT\*

Marita Duecker<sup>†</sup>, Georg Lehrenfeld<sup>‡</sup>, Wolfgang Mueller<sup>†</sup>, Christoph Tahedl<sup>‡</sup>

<sup>†</sup> C-LAB

<sup>‡</sup>Heinz-Nixdorf Institut

Fuerstenallee 11, 33102 Paderborn, Germany

## ABSTRACT

Understanding and debugging of concurrent systems is a critical activity in systems design. Means supporting the visual inspection of drawings as well as the animation of the execution are expected to play a significant role in future. Since Pictorial Janus (PJ) has been defined as a completely visual language including an advanced animation semantics it serves well as a high-level language for the debugging and rapid development of concurrent systems. In this article, we present an interactive PJ specification and analysis environment for the rapid prototyping of concurrent systems. The environment covers an editor with integrated animation facilities and an interpreter enhanced by debugging facilities.

## INTRODUCTION

The specification and analysis of distributed systems is an error-prone activity. Even if the designer exercises every care there is a strong probability for introducing functional and timing errors during the early design phase. The complexity of distributed systems can hardly be managed within the classical programming environment. The design process of complex systems is only manageable by advanced design tools. A recent trend in high-level electronic systems design indicates the use of advanced graphical design tools (Gajski et al. 1994). Visualization and animation tools supporting the analysis and execution of concurrent programs are expected to play a most important role in the next years (Kraemer and Stasko 1993; De Pauw et al. 1994). Most of the applied and known graphical means provide a static representation of the dynamics of programs. Only little attention has been paid to the animation of concurrent programs in order to achieve a better and quicker understanding of their execution (De Pauw et al. 1994).

Pictorial Janus (PJ), introduced by Kahn and Saraswat in (Kahn and Saraswat 1990), is the complete visual representation of the concurrent logical programming language Janus covering an advanced animation semantics. PJ was the first representative of a programming language with a complete visual representation providing a smooth animation of the program's execution. This article introduces a distributed PJ specification and analysis environment for the rapid prototyping of concurrent systems. The system covers an editor with integrated animation facilities and an interpreter enhanced by debugging facilities. By the tight integration of the animator with the editor the user may interact with the program during the animation. It is even possible to modify the program during the animation without halting the execution.

The remainder of this paper is organized as follows. We first discuss related work in visual representation and animation. Section 2 investigates related work. In Section 3, we present PJ. Section 4 introduces our distributed system for the specification and visual analysis of PJ programs and discusses problems when analyzing concurrent systems. The paper closes with a conclusion and outlook.

## RELATED WORK

Graphical symbols have a long tradition in the design of digital circuits at switch, gate, and RT level (transistors, gates, alus, etc.). At the algorithm level the visualization of the design entities is not prescribed by any standardized or commonly agreed symbols. Today's popular means for system design are mainly control flow oriented combined with data flow graphs (Control/Data Flow Graph), first order logic (Predicate Transition Nets), or programming languages (StateCharts, SpecCharts, Speedcharts<sup>1</sup>, VisualHDL<sup>2</sup>, etc.). (Gajski et al. 1994). Systems visualize the order of execution, data dependencies, state transitions, or processes connected by communication links by the means of circles, (embedded) boxes, and links between them. Several systems incorporate pixmaps for various purposes of graphical application specific representations. vVHDL, for instance, provides an iconic representation of the sequential VHDL statements (Miller-Karlow and Golin 1992).

Some of these systems also provide very basic animation facilities by blinking or changing the color of the currently active symbol or statement. Various frameworks are available for the advanced animation of concurrent systems. An overview can be found in (Kraemer and Stasko 1993). Advanced concepts are mainly based on algorithm animation (Brown 1988). For algorithm animation the program code is annotated by functions which implement the animation. Recent approaches introduce graphical captures for the specification of animation objects and the integration of operations into a program (Carlson and Burnett 1995).

In contrast to the currently popular means for system specification, Pictorial Janus (PJ) provides the complete visualization of the control flow and data flow without any textual means. This permits a visual inspection of both flows without combining two separated drawings or retrieving the data dependencies from the textual program code. In contrast to the approaches based on algorithm animation, PJ unifies the means for defining the program and the animation. When drawing the program the user assigns the geometry and the layout for the animation to the individual objects. Additionally, PJ provides a smooth continuous animation of its execution by continuous motions and the morphing of individual objects—not just blinking or changing the color.

## PICTORIAL JANUS

PictorialJanus (PJ) is a complete visual programming language based on the parallel logical programming language Janus introduced by Kahn and Saraswat in (Kahn and Saraswat 1990).

The basic elements of a PJ program are graphical primitives, i.e., closed contours and connections. The meaning of a closed contour is independent from its geometrical representation and graphical context, i.e., shape, size, color, etc. The basic primitives are combined to objects by topological relationships (attachment and inclusion). For objects, PJ dis-

\*Published in the Proceedings of the 10th European Simulation Multiconference, June 2-6, Budapest, Hungary.

<sup>1</sup>SPeeDCHART is a trademark of SPEED SA.

<sup>2</sup>VisualHDL is a trademark of Summit Design, Inc.

tinguishes *agents*, *functions*, *relations*, and *messages*. Each object may have *ports* in order to establish a connection to other objects. Figure 1 gives a list of the various PJ objects. In that figure ports are filled grey in order to emphasize their contours.

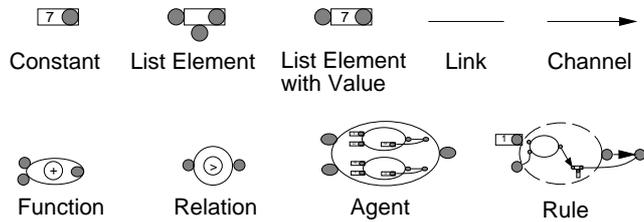


Figure 1: PJ Objects

*Constants* and *list elements* are denoted as messages. List elements establish more complex data structures. Constants and list elements may hold values. Different elements may be connected by *links*, which are represented by undirected lines. Links represent data dependencies.

Functions and agents consume and produce messages. An agent is defined by a closed contour with a set of external (argument) ports. The behavior of an agent is defined by a set of *rules* which are located inside its contour. A rule is basically a copy of the agent's interface (contour and ports). Each rule defines the behavior of an agent with respect to different input patterns (*guards*). The relation objects are used to define constraints between messages within a guard. The guards are located outside the rule's contour whereas the behavior (*subconfiguration*) is defined inside its contour. The subconfiguration defines a set of linked objects, i.e., messages, functions, and agents, being created when matching the rule. Instead of explicitly specifying the behavior of an agent, a *call arrow* may instantiate another agent or the agent itself, recursively. A recursive call makes agents persistent by replacing an agent by an instance of itself. A function has a predefined behavior denoted by the symbol inside its contours. Channels establish directed connections between two external ports. Their intuitive meaning is to "send" messages to other agents or functions.

For the specification of real-time systems we have extended PJ by the concept of timers<sup>3</sup> A timer is a unary function which when being created start a timeout. The duration  $t_i$  of its timeout is given inside its contour. Timers are controlled by a global clock which is denoted as the current simulation time  $T_c$ . Thus, when being created a timer  $t_i$  expires at  $T_c + t_i$ . The Timed PJ execution cycle is sketched by the computation steps given in the following algorithm.

```

WHILE true DO
  FOR each agent/function DO
    IF input event THEN
      resume and execute agent/function;
    ELSE
      update current time;
      resume and execute expired timers;
    FI;
  OD;
OD;

```

Timers execute on the advance of the current time. The above algorithm advances the time if no other events have oc-

<sup>3</sup>The concept of a timer is comparable to a timer of the CCITT standard specification language SDL (Lehrenfeld et al. 1995). The simulation cycle is basically inherited from the IEEE standard hardware description language VHDL (IEEE 1994).

curred at any agent or function. In that case the current time is updated to the time the next timer is expected to expire. All timers which have reached their expiration by the update of the current time are resumed and executed, i.e., they are replaced by a link and an event is sent to the agent/function at their output.

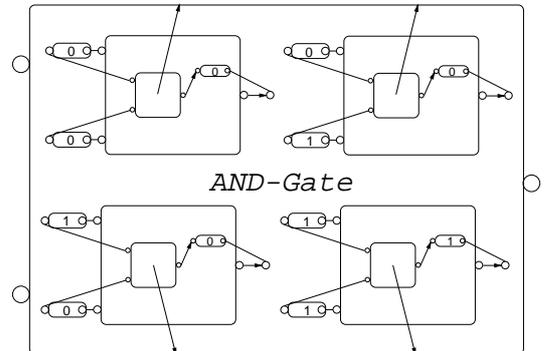


Figure 2: PJ Agent with 4 Rules

We briefly outline PJ by a simple example from the electronic modeling domain defining an AND-gate (see Figure 2). An AND-gate can be defined as a function mapping input values to an output value. Since the gate has two inputs there are four different 0/1 input combinations. The various input patterns are modeled as guards of four different rules—one rule for each combination. In all cases a new subconfiguration with an output value (message) as well as the recursive replacement of the agent is generated.

Whereas the PJ drawing is a program the execution is the animation. Due to their rotational and dilational invariance PJ objects may vary in size and outlook during the animation. When the animation advances, valid PJ follow-up configurations are derived from the current configuration. The in-between of the configurations is defined by a smooth morphing of the individual objects. We distinguish the animation at *system level* and at *component level*.

The system level animation views the agents as black boxes, i.e., the agents and the scheduling and consumption of messages is animated. The component level animation additionally animates the pattern matching, the selection, instantiation, and linking of a subconfiguration. The component level animation of an agent is given by the following steps.

1. *Instantiate Subconfiguration*
2. *Enlarge Selected Rule*
3. *Delete Agent, Matched Objects, Rule, and Guard*
4. *Enlarge Subconfiguration*
5. *Shrink Links*

Figure 3 gives a short animation sequence by six snapshots when matching a 0-0 combination at the input w.r.t. the definition of the AND-Gate in Figure 2. Frame 1 shows the start configuration with values scheduled at the input of the gate and the output connected to a receiving agent.

## A DISTRIBUTED SYSTEM ARCHITECTURE FOR VISUAL ANALYSIS

Our system for the interactive specification and debugging of PJ can be roughly divided into two main components: an integrated editor/ animator and a (possibly) distributed interpreter. Figure 4 gives an overview of the system's architecture. The system is based on a tight integration of the editor and animator since the processing and visualization of the graphics require high frame rates. For a distributed environment we have decided to decouple interpreter and animator

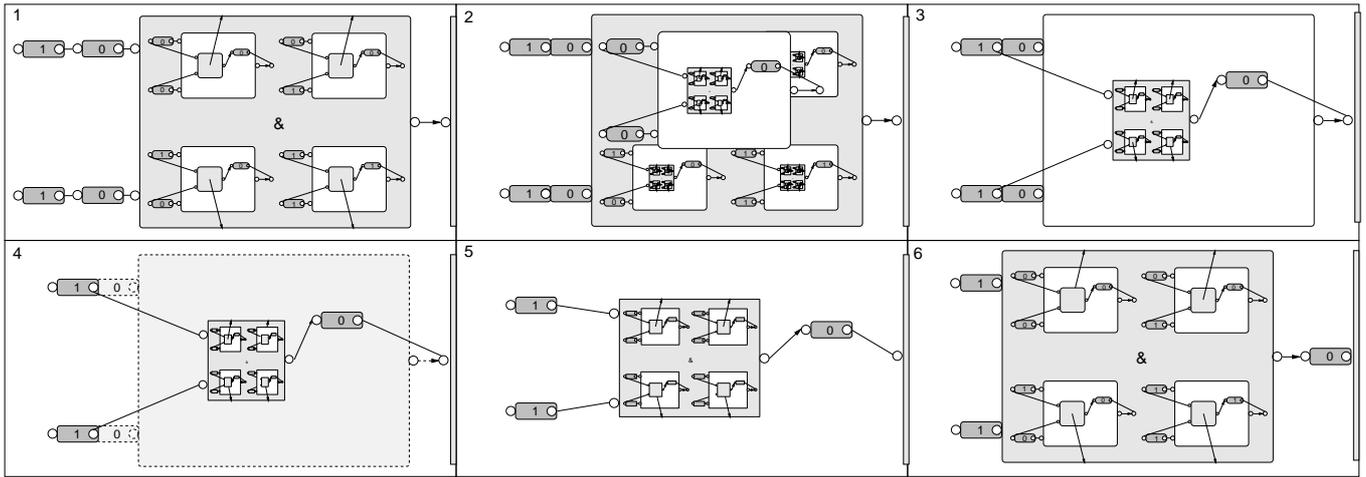


Figure 3: Component Level Animation

since due to our abstract definition of the animation interface their interprocess communication is not a major bottleneck in the performance of the system. This separation supports a concurrent control of the animator by a parallel execution as a set of concurrently running processes.

The system has been developed in C++ with the editor toolkit Eos (Kaufmann et al. 1994) under SunOS 4.X. It additionally runs under IRIX, Solaris, and WindowsNT. The process communication is implemented by the use of PVM 3.3. We have implemented a sequentialized execution in order to facilitate a first system debugging. The implementation of a concurrent interpretation is currently under investigation. Figure 5 shows a screenshot of PJ Editor/Animator and Interpreter with debugging interface.

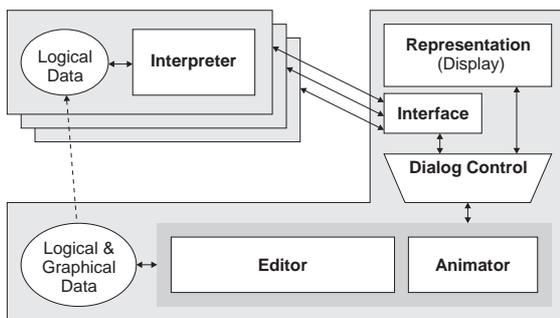


Figure 4: System Architecture

### Editor/Animator

**The editor** is a graphical entry for modifying the static representation (drawing) of the language. Basic functions are the creation, deletion, and modification of individual objects as well as basic navigation facilities, e.g., scrolling.

**The animator** computes the visualization of the execution, i.e., it processes the animation commands received from the interpreter. The animator modifies graphical objects w.r.t. their (topo)logical relationships. Smooth transitions are provided by continuous motions, morphing, and scaling of the individual objects. Our implementation supports simultaneous animation, e.g., it displays the motion of multiple objects.

Editor and animator both modify a shared set of graphical objects. The simultaneous access to these objects is realized by the concept of non-blocking commands introduced by Kaufmann in (Kaufmann 1994) and implemented by the editor toolkit EOS (Kaufmann et al. 1994). Non-blocking commands basically compare to cooperative threads which are serialized by a scheduler. The animator and the editor are integrated as non-blocking commands receiving requests from the interpreter(s) and events from the user interface, e.g., mouse clicks. Requests trigger the manipulation of graphical objects in order to generate the individual frames of the animation. A timer is started after the elaboration of requests which are received from the interpreter(s). Then the program control is passed to the editor command which computes the currently available events from the user interface until the timer expires. The execution of requests from the interpreter continues after the expiration of the timer enabling the animator command again. This technique provides a pseudo-simultaneous execution of the editor and animator.

### Interpreter/Debugger

The interpreter executes the program on a logical level, i.e., does not compute geometrical data such as coordinates but it only computes the logical structure (logical data) of the program triggering the animator with abstract animation commands, e.g., "match rule  $R_2$  of agent  $A_6$ ", "morph object  $A_1$  to object  $A_2$ ", or "rescale object  $A$  by a factor of 2". The interpreter can be basically used as a debugger by introducing control points to the interpretation cycle.

Decoupling animator and interpreter as autonomous processes requires logical and temporal synchronization. When starting the animation the data structures of the interpreter are automatically being initialized by the program. Whenever the graphics (programs) are being modified during the execution by the editor the interpreter is notified in order to update its logical data. An interpreter invokes an animation operation by a unique operation identifier. The animator notifies its (temporal) completion by sending the corresponding commit signal.

### SEQUENTIAL VS. CONCURRENT EXECUTION

Concurrent programs can be analyzed by a sequential(ized) and a concurrent execution.

## Sequential Execution

For the first analysis of the basic protocols of a system it is sufficient to investigate only one possible sequence in the execution of a concurrent program. The sequentialized implementation additionally eases the implementation of debugging facilities since it abstracts from considerable problems in the debugging of concurrent system as being discussed in the next paragraph. For the sequential execution most concepts can be inherited from the well-known debuggers for textual languages, e.g., dbx and gdb. The complete visual representation of PJ simplifies the debugging in many cases. The following gives a brief overview of the implemented functions.

PJ objects can be deleted, added, and modified at any time during the run-time (animation) even without halting the program. User controlled execution facilitate the step-wise sequential execution of PJ agents. An explicit rule selection allows to alter the execution of a program. Rules can be selected in order to influence the non-deterministic choice of rules with the same guards. Agents can be selected in order to change their order of execution.

The program's execution can be manually interrupted by the user at any time or automatically by setting breakpoints. Breakpoints refer to PJ's syntactical elements, e.g. agents, rules and messages. Objects are being selected through the editor. The debugger is automatically notified by their selection. Agents and rules may inherit breakpoints to the objects in the newly created subconfiguration. During the execution of the program single configurations of an animation can be recorded by setting bookmarks. This facilitates the complete recording of an execution and its analysis. Profiling supplies basic statistics recording the number of agent calls and chosen rules, etc. w.r.t. the number and types of agents as well as the number of messages with their recipients.

The debugger allows to change between different presentations of a PJ program. The system view displays an agent as a black box only animating the generation and consumption of messages. The component view animates the pattern matching in addition.

## Concurrent Execution

Debugging programs in a parallel execution requires advanced methods and raises considerable problems. Due to unexpected delays in the communication the program's behavior can differ from run to run. This makes errors difficult to detect and to eliminate. In the remainder of this paragraph some of the main problems as introduced by (Alabau et al. 1993; Leblanc 1994) are discussed.

*Naming.* Naming refers to the ability of selecting an object in a parallel application in order to perform operations on the selected object. In the case of multiple instances of an object its identifier appears no longer to be unique. A solution to that is given by graphical tools visualizing individual objects where objects can be selected directly within a drawing. In our case we can ignore that problem since PJ provides a complete visual representation.

*Global State.* The global state of a concurrent program is composed of the local states of the individual processes as well as the states of all communication channels. The global state is required for detecting deadlocks and program termination. A centralized simulator controlling the distributed execution can be one solution to that problem. Another solution provide algorithms (Chandy and Lamport 1985; Miller and Choi 1988) computing the global state by taking distributed snapshots.

*Breakpoints.* Breakpoints are generally used to interrupt the execution of a program in order to analyze the current state. In sequential programs, a breakpoint results in the im-

mediate interrupt of the execution. In distributed programs, however, it is not guaranteed that all components of a program immediately stop their execution since in case of reaching a breakpoint in one process all other processes have to be notified by the occurrence of that breakpoint. The other processes are stopped with a certain delay, i.e., the state when the system stops is different from the state when the breakpoint has occurred. Three different methods were developed to solve that problem. *Local breakpoints* are used to analyze only the state of one process without considering the remaining processes of the program. *Global snapshots* allow a consistent view of the global state of a program and therefore simulate global breakpoints. *Causal Distributed Breakpoints* are used to record events and other matters for a particular state. When reaching a breakpoint all other processes are notified and their states are restored to the state where the breakpoint has occurred.

*Replay.* Concurrent programs are often non-deterministic. For the debugging it is essential to reproduce a particular program sequence. A *replay* is typically subdivided into two phases: the recording and the replay. The replay is typically realized by using message protocols recording the arrival of messages.

For the concurrent execution of PJ programs the program is partitioned into a set of interpreters, e.g., one interpreter for each agent can be an adequate solution. The interpreters concurrently trigger the animator. Messages are represented as signals being exchanged between the individual interpreters. Message queues are implemented as signal buffers. The concurrent execution refers to an uncontrolled execution of the individual interpreters. By stepping through the execution each agent is being triggered once in order to check for a possible application of rules. In the next step the selected rules are executed before the agents are triggered again. That execution applies a synchronous execution to the concurrent program since all agents executed the selected rules synchronously. Each step assigns a global state to the program which is being stored as a PJ keyframe for the purpose of a replay. That keyframe then refers to a global snapshot. By that synchronous execution we avoid all of the above problems, like global state, breakpoints, and replay. Naming is solved by PJ anyway. Breakpoints for an unsynchronous parallel execution is not a subject of our present investigations.

## CONCLUSIONS AND FUTURE WORK

We think that PJ and its animation provides great means to understand an executable specification and to detect errors in the basic communication during the first prototyping of a concurrent system. In particular, the complete visualization at component level greatly helps through the visual inspection of a program. All objects can be directly accessed through the editor for debugging purposes which eases their selection, e.g., for tracing or setting breakpoints. Facilities gained by the complete visualization and animation, e.g., record, replay, additionally help to understand the execution is further details.

Other articles have shown the applicability of PJ in the fields of high level SDL protocols (Lehrenfeld et al. 1995) and low level bus protocols (Mueller et al. 1995). For applications in process modeling and manufacturing process control we have extended PJ by external functions which permit the integration of existing software and hardware. All these examples indicate much better analysis facilities compared to the classical approach using a set of concurrently running sequential debuggers on a textual basis.

Nevertheless, we also found some PJ limitations. Inherently sequential problems as well as an imperative programming style result in very inefficient PJ representations. Imple-

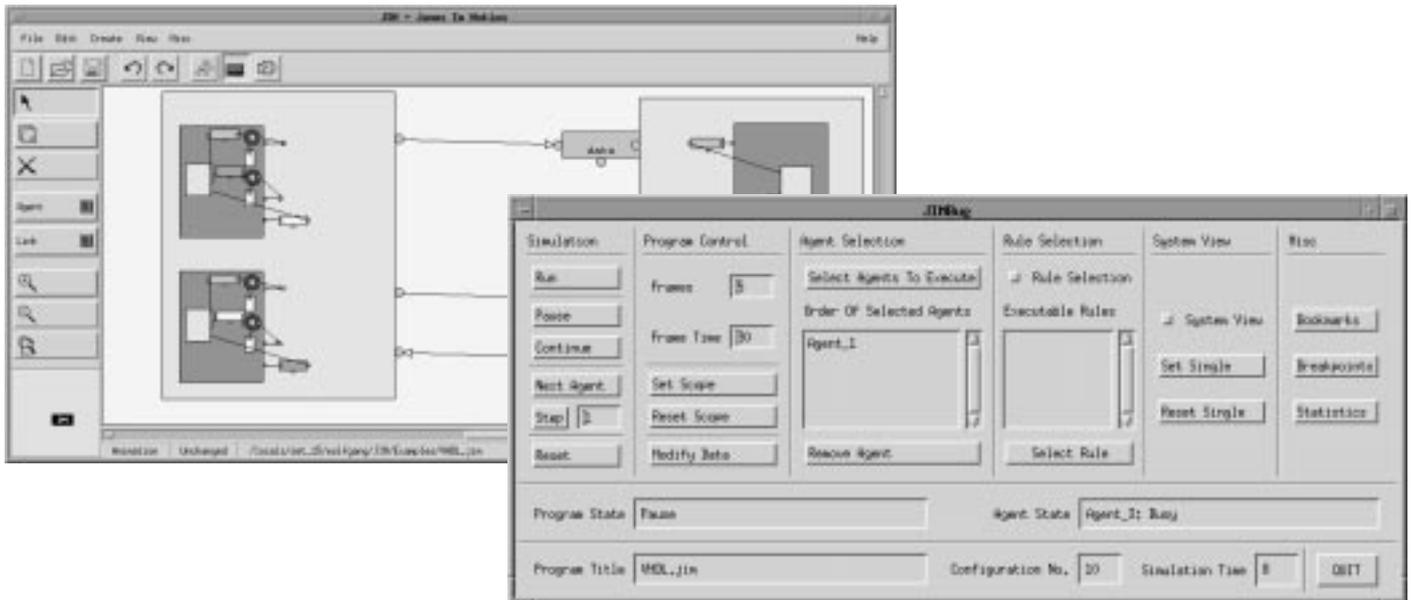


Figure 5: Screenshot PJ Editor/Animator and Interpreter with Debugging Facilities

menting these parts by a sequential imperative programming language and integrating them as external functions overcome most of these problems. However, our present implementation lacks additional tools in order to efficiently support the interfacing of external functions. Our current investigations concern hierarchical concepts for rules. E.g., agents with a considerable set of rules become hard to analyze by visual inspection. The hierarchical grouping of rules w.r.t. related guards would help us in many cases when applying PJ to complex systems specification.

Our experiences with the presented interactive animation system has shown that the tight integration of animator and editor as well as the decoupling from the interpreter has major advantages. Firstly, it provides ideal rapid prototyping facilities since a program drawn by the editor can be immediately animated. The drawing can be modified even without halting the animation. Secondly, the separation of the system into editor/animator and interpreter allows to easily replace the individual system components. That allows us to apply our animator as a visualization tool for other interpreters or simulators. In addition, as it is shown by our most recent investigations, the interpreter can also be used to trigger an OpenInventor-based 3-D visualization.

## ACKNOWLEDGEMENTS

We would like to thank Hermann-Josef Kaufmann for his great help in his EOS support as well as Thomas Kern and Taner Cinkoese for their Windows NT support. Uschi Hudson and F.J. Rammig have reviewed this work.

## REFERENCES

Alabau, M. et al. 1993. "A programming environment dedicated to a model of explicit parallelism". In *Environments and tools for parallel scientific computing*, J.J. Dongarra and B. Tourancheau, eds., Elsevier Science Publishers B. V.

Brown, M.H. 1988. *Algorithm Animation*. MIT Press, Cambridge, MA.

Carlson, P. and M.M. Burnett. 1995. "A Seamless Integration of Algorithm Animation into a Visual Programming Language with One-Way Constraints". In *International Workshop on Constraints for Graphics and Visualization*. Cassis, France.

Chandy, K.M. and L. Lamport. 1985. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems*, 3(1):63-75.

De Pauw, W., D. Kimelman, and J. Vlissides. 1994. "Modeling Object-Oriented Program Execution". In *Eighth European Conference on Object-Oriented Programming*, Bologna, Italy. Springer-Verlag, Berlin.

Gajski, D.D., F. Vahid, S. Narayan, and J. Gong. 1994. *Specification and design of Embedded Systems*. Prentice Hall, Englewood Cliffs, NJ.

IEEE. 1994. New York, NY, USA. *IEEE Standard VHDL Language Reference Manual-IEEE Std 1076-1993*.

Kahn, K. and V.A. Saraswat. 1990. "Complete Visualizations of Concurrent Programs and their Executions". In *1990 IEEE Workshop on Visual Languages*. Skokie, IL.

Kaufmann, H.-J. 1994. "EDIS: Eine objekt-orientierte Software-Architektur fuer graphische Editoren". PhD thesis. Paderborn University, Germany.

Kaufmann, H.-J., Th. Kern, and R. Zhao. 1994. "Detailed Functional Specification EOS 1.0". Technical Report BT-HCI 94. Cadlab, Paderborn, Germany.

Kraemer, E. and J.T. Stasko. 1993. "The visualization of Parallel Systems: An Overview". *Journal of Parallel and Distributed Computing*, vol. 18, Academic Press Inc., Dordrecht, Netherlands.

Leblanc, Th.J. 1994. "Debugging in Distributed Systems". In *Encyclopedia of Software Engineering*, John J. Marciniak, ed., John Wiley & Sons, Inc.

Lehrenfeld G., and W. Mueller, and C. Tahedl. 1995. "Transforming SDL into a Complete Visual Representation". In *Proceedings of the 1995 IEEE Symposium on Visual Languages*. Darmstadt, Germany.

Miller, B.P. and J.D. Choi. 1988. "Breakpoints and Halting in Distributed Programs". In *Proceedings 8th International Conference on Distributed Computing Systems*, 316-323.

Miller-Karlow, D.L. and E.J. Golin. 1992. "vVHDL: A Visual Hardware Description Language". In *Proceedings of the 1992 IEEE Workshop on Visual Languages*. IEEE CS Press, Los Alamitos, CA.

Mueller W., G. Lehrenfeld, and C. Tahedl. 1995. "Complete Visualization and Animation of Protocols". In *Computer Hardware Description Languages & Application*. Tokyo, Japan.