A Generic System for Interactive Real–Time Animation

Marita Duecker[**], Georg Lehrenfeld[*], Wolfgang Mueller[**], Christoph Tahedl[*]
[*] Heinz Nixdorf Institut, Fuerstenallee 11, 33102 Paderborn, Germany
[**] C–LAB, Fuerstenallee 11, 33102 Paderborn, Germany

Abstract

We present a new approach to an interactive design and analysis environment for visual languages. The main components, i.e., editor, animator, and interpreter, are introduced. Their interactions are being investigated emphasizing the interpreter–animator interaction and defining an interface supporting different levels of automation. The interpreter performs the executions on a logical level and triggers the animator. The interactive animation provides a very high degree in liveness since it is based on the tight integration of the animator and editor. The proposed architecture permits the distributed implementation of a system for real–time animation. Our concepts are validated by the implementation of a debugging environment for the complete visual programming language Pictorial Janus.

1. 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 [8]. Visualization and animation tools supporting the analysis and execution of concurrent programs are expected to play a most important role in the next years [12, 7]. Most of the applied and known graphical means provide a statical 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 [7]. With the availability of multimedia platforms and the increasing power in graphical processing 2–D animation will become most important since it greatly helps in understanding the execution of complex systems. Current applications of 2–D animation mainly concern computer–based training and education [6] as well as the analysis of complex dynamic systems [16].

Several systems only provide very basic animation with shuffling bitmaps or moving rectangles between objects. Other systems perform an off–line generation of the individual frames which can only be viewed but not manipulated. We investigate real–time animation applicable for interactive debugging and prototyping of visual languages. Though the approach is demonstrated by the example of the completely visual general purpose programming language Pictorial Janus it should be applicable for arbitrary languages supporting the concept of concurrently running agents exchanging messages. Due to Tanimoto's classification of interactive environments [19] we achieve the highest level of liveness. This high degree of user–system interaction is achieved by a tight integration of the animator and the editor. The animator basically translates interpreter commands into motion manipulating graphical objects. The editor has simultaneous non–blocking access to the graphical objects allowing the modification of the graphics during the animation even without halting the execution. For the communication of concurrently running agents the animation supports the simultaneous motions of the individual objects when being triggered by a set of parallel running interpreters. For this, we have defined a clear separation of the graphical processing and the interpretation based on a detailed investigation of the animator–interpreter communication which allows a distributed implementation. The animation interface is defined along the classification of 3–D animation in [13]. In order to demonstrate the applicability of our concepts we have chosen the complete visual programming language Pictorial Janus since due to its close relation to concurrent Prolog it is well applicable to rapid prototyping of concurrent systems where animation provides additional value during the first functional analysis.

The remainder of this article is structured as follows. The next section discusses related work. Section 3 introduces Pictorial Janus (PJ) which is taken as our example. We introduce PJ's execution model and advanced animation model. Section 4 summerizes the classification of (3–D) animation systems. Section 5 introduces the individual components of an animation system, their interaction, and gives an overview of the implementation. The article closes with a conclusion.

2. Related Works

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[1], VisualHDL[2], etc.) [8]. 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 [15].

Some of these systems provide very basic animation by blinking or changing the color of the currently active symbol or statement. These facilities give useful insights to the current state of the (sequentialized) simulator but do not provide facilities for a deeper analysis of the I/O behavior of parallel processes. In the remainder of this article we understand animation as motion and transformations of objects. Investigations have shown that smooth state transitions make animation easily traceable [5]. Considerable work in the visualization of dynamics has been done in the field of *algorithm animation* mainly based on the work of Brown [3]. Whereas Brown's BALSA is based on a textual system recent research presented by Burnett et al. [4] assigns algorithm animation to a visual programming language (Forms/3) applying Stasko's path/transition paradigm [18], i.e., defining the object, its path, the transition type (e.g., intensity, visibility, color), and a set of constraints. All these approaches still provide two different means when modeling an animation. On the one hand the user has to manage the semantics of the programming language. On the other hand she has to deal with the semantics of visual means. Completely visual programming languages can overcome these difficulties as it is demonstrated by Kahn's and Saraswat's Pictorial Janus (PJ) [17, 9]. PJ's complete dynamic semantics is defined by a set of graphical rewrite rules. The execution can be relative easily derived from those rules.

3. Pictorial Janus

Pictorial Janus (PJ) is a complete visual programming language based on the parallel logical programming language Janus introduced by Kahn and Saraswat in [9].

[1] SPeeDCHART is a trademark of SPEED SA.
[2] VisualHDL is a trademark of Summit Design, Inc.

3.1. PJ Objects and Programs

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 distinguishes *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.

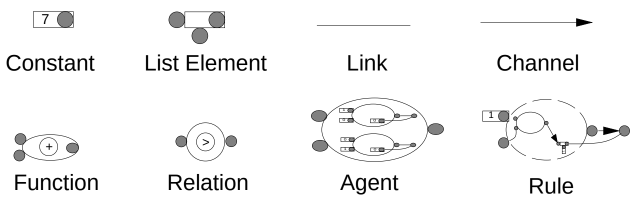

Figure 1. PJ Objects

Constants and *list elements* are denoted as messages. Constants hold values. List elements establish more complex data structures. 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 contour. *Channels* establish directed connections between two external ports. Their intuitive meaning is to "send" messages to other agents or functions. For a comprehensive representation of PJ programs two short cuts are introduced in Figure 2.

We briefly outline PJ by a simple example from the electronic modeling domain defining an AND-gate (see Figure 3). An AND-gate can be defined as a function mapping

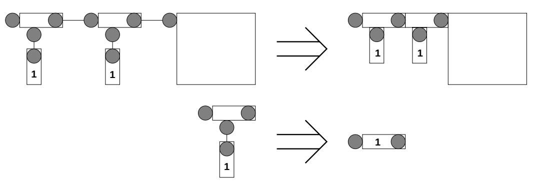

Figure 2. PJ Short Cuts

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.

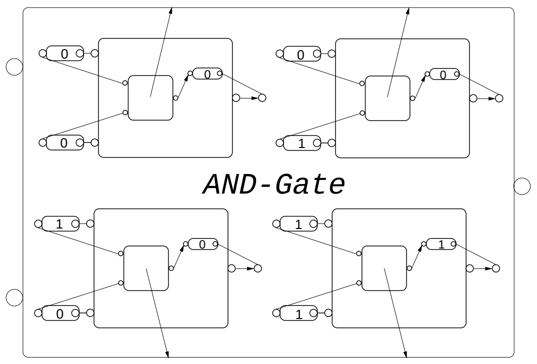

Figure 3. PJ Agent with 4 Rules

3.2. Execution

A PJ program defines a net of communicating agents and functions concurrently sending and receiving messages. In the case that an event has been detected at any external port of an agent/function, i.e., when any message is being scheduled, the agent/function resumes and checks whether any rule can be successfully applied by performing a pattern matching. The basic PJ execution can be sketched by the following simplified steps.

On the arrival of at least one new message at one external port of agent A[3]

1. *Check Rules:*
   A resumes and checks each rule r for matching its guards with the objects at the external ports of A (messages and functions). Constraints given by relations are checked if being defined.
2. *Select Rule:*
   In the case that any rule r matches r is a candidate for the further computation of the agent. One rule r is nondeterministically selected from the set of candidates.
3. *Create Subconfiguration:*
   The subconfiguration defined by r is generated. Newly suspended agents/functions are suspended by default.
4. *Link Subconfiguration:*
   Objects of the subconfiguration are linked to the existing configuration. As a result new messages are scheduled to agents/function and generate events.
5. *Delete Agent and Guards:*
   A, its guards, the matched input objects including their connections are finally deleted.

[3] Note, that functions can be viewed as special forms of agents and are thus covered when investigating only agents in the remainder of this article.

3.3. Animation

Whereas the PJ drawing is the static representation of a program the execution is performed in terms of an advanced animation. During the animation 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. This is possible since due to their rotational and dilational invariance PJ objects may vary in size and outlook during the animation. We distinguish *system level* and *component level* animation.

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

1. *Instantiate Subconfiguration:*
   The objects within a subconfiguration of the selected rule are instantiated. Call arrows of agents are replaced by the corresponding behavior.
2. *Enlarge Selected Rule:*
   Once a rule r of A has been selected r continuously grows until its contour overlaps the agent's contour and the guards overlap the corresponding input objects. The visualization of the pattern matching is performed by a morphing of the finally overlapping objects.
3. *Delete Agent, Matched Objects, Rule, and Guard:*
   The matched as well as the matching objects smoothly disappear.
4. *Resize Subconfiguration and Shrink Links:*
   The subconfiguration is resized until it fits to the space of previously deleted objects. The shrinking of links finally animates the motion of messages to their destination.

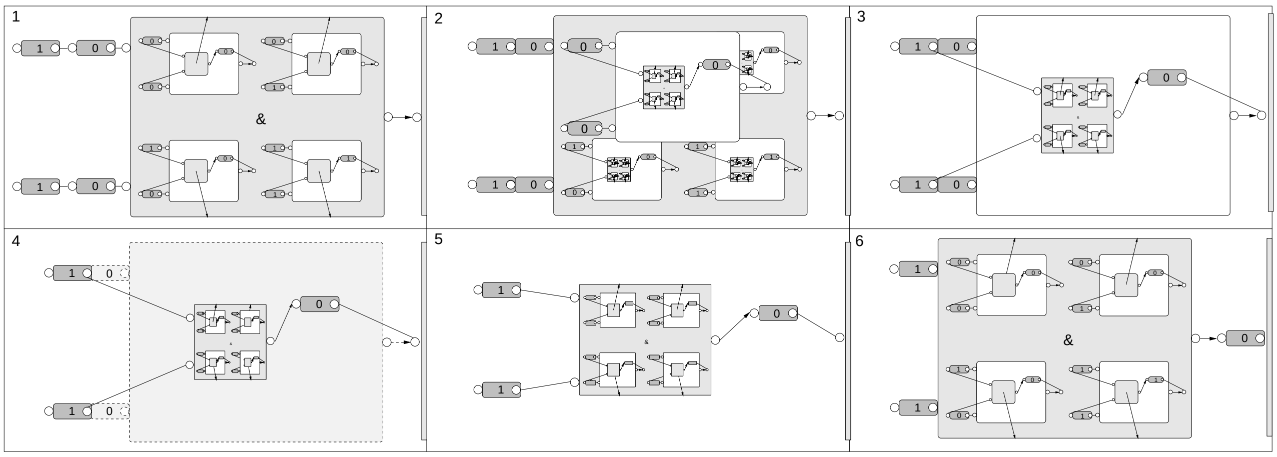
Figure 4. Component Level Animation

Figure 4 gives a short animation sequence by six snapshots when matching a 0–0 combination at the input with respect to the definition of the AND gate in Figure 3. Frame 1 shows the start configuration with values scheduled at the input of the gate and the output connected to a receiving agent.

4. Animation Classification and Techniques

Classifications of 3–D animation are given by Magnenat-Thalmann and Thalmann in [13, 14]. Though these classifications are introduced for 3–D animation they are well applicable to 2–D as it is shown in Section 5.

A first classification distinguishes *computer-aided* and *modeled* animation [13]. In modeled animation the motion is specified by an algorithm or a function. Computer–aided animation is based on the interpolation between keyframes.

A further classification refers to the process of generating the individual frames: *frame–by–frame* and *real–time* animation [13]. As an animation should be recorded with at least 18–24 frames per second (fps) an on–line or real–time animation is often not possible, in particular in the field of 3–D animation with complex rendering including illumination effects. In those cases frames have to be off–line generated.

A classification of animation systems with respect to their level of automation can be found in [13]. The following gives a brief overview and presents techniques applied at the individual levels [14].

Level 1. Frames are drawn by a graphical editor. The editor provides special features for manually transforming the individual objects. The animation player accepts the individual drawings as frames and records them in a predefined sequence.

Level 2. The in–betweens of given (key)frames are computed by interpolation. Motions follow a specified path. In–betweens are computed by applying *keyframe animation* which is also known as *in–between* technique. The user specifies different key drawings. In–between drawings are generated by computing linear distances or a spline interpolation between given points. Linear interpolation typically generates synthetic motions where the spline interpolation generates smooth motions. *Parametric keyframe animation* is a variant of the keyframe animation. Attributes of objects are controlled by specific parameters, e.g., size, position, angle, and illumination. The in–between frames are computed by interpolating the parameters.

Level 3. Whereas previous levels define *how* objects are animated Level 3 and Level 4 define *what* is animated. Objects are automatically transformed and rotated. The camera position can be changed virtually. At this level we distinguish *algorithmic* and *task level animation*. In algorithmic animation motion is described as a sequence of transformations, e.g., rotation or translation. Every transformation is controlled by parameters. Modifications of parameters can be restricted by physical laws. For task level animation motion is described with abstract tasks that refer to objects. The animation system transforms the abstract tasks into concrete motion. Typical tasks are: “move object from location A to location B” or “move lips for speaking a sentence”.

Level 4. *Actors*, i.e., the objects that are to be animated, perform the animation by themselves. Small procedures as well as graphical constraints are assigned to actors as methods. Methods are triggered before computing the next frame in order to calculate new attributes of the actor. An actor–based animation typically defines a (concurrent) network of communicating actors communicating by exchanging information.

Our system performs a 2–D computer–aided real–time animation. The animation interface has been designed along the above levels of automation providing operations for parametric keyframe animation, algorithmic animation, task level animation, and actor–based animation.

5. The Animation System

An animation system for a complete visual representation can be roughly divided into three main components.

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

The interpreter executes the program on a logical level. The interpreter is not in charge of computing geometrical data such as coordinates. The interpreter computes the logical structure (logical data) of the program and triggers 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 animator computes the visualization of the execution, i.e., it processes the animation commands received from the interpreter and transforms them to the manipulation of graphical objects given as a set of graphical data structured with respect to their (topo)logical relationships. Smooth transitions are provided by continuous motions, morphing, and scaling of the individual objects. Advanced implementations should support simultaneous motions, i.e., the elaboration of multiple animation operations at the same time. That means, the animator checks for further operations after each frame.

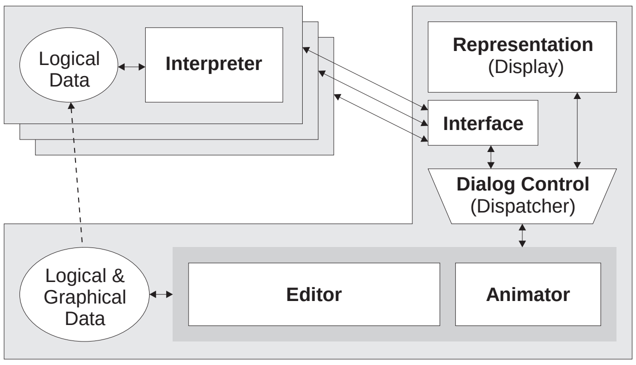

Figure 5. System Architecture

For a distributed environment we have decided to decouple interpreter and animator since due to our abstract definition of the animation interface their interprocess communication is not a major bottleneck in the performance of the system. The main bottleneck is the processing and visualization of the graphics since smooth motions require high frame rates. Therefore, the system is based on a tight integration of the editor and animator as depicted by Figure 5 and outlined in the remainder of this section. This separation supports a concurrent control of the animator by a parallel execution as a set of concurrently running processes. It additionally gives us a considerable flexibility coupling the animation with other interpreters and thus experimenting with the animator as a visualization tool of PJ related execution models.

5.1. Editor–Animator Interaction

Editor and animator both manipulate graphical data. The simultaneous access to this common data structure is solved by the concept of non–blocking commands introduced by Kaufmann in [10] and implemented by the editor framework EOS in [11] (cf. Subsection 6). The basic concept is the conceptual decomposition of interactive programs in presentation, dialogue control, and semantics (logical and graphical data). The presentation manages the visual representation to the end–user. The internal representation of a graphical entity (window, drawing area, button, text field, etc.) is denoted as a device. The dialogue control dispatches input events (mouse clicks, etc.) received from the devices and activates the corresponding command. That way, the dispatcher performs a serialization of input requests. A command is responsible for the interpretation of the input event, the manipulation of the graphical objects, and the notification of devices in order to perform a redraw, etc.

An animator can be easily integrated to this structure as a command receiving requests from the interpreter through the dispatcher (cf. Figure 5). If no input event is available the animator starts a timer and releases the program control until the next frame. When the timer expires an input event is generated which invokes an animator command. Since commands are non–blocking animator and editor commands can be invoked at the same time. This concept permits a manipulation of the drawing by the editor while the animator is active and thus provides a maximum degree of interactity with the user.

5.2. Animator–Interpreter Interaction

Decoupling animator and interpreter as autonomous processes requires logical and temporal synchronization. We first introduce the basic synchronization principles and then further present a flexible interface for interpreter–animator communication.

5.2.1. Synchronization

In a distributed environment the interpreter and the animator have to manage their own representation of the program as it is shown in Figure 5. In order to distinguish both data structures we denote them as logical data and graphical data. We further distinguish run–time and static objects. Run–time objects are global objects which are manipulated during the execution of a program, e.g., agents, functions, and messages. They are identified by dynamic object identifiers. Considering PJ run–time objects are agents, functions, and messages being scheduled to agents or functions. Static objects are local to run–time objects and are not dynamically manipulable. They are identified by their static symbolic names. In PJ, for instance, static objects are objects enclosed by an agents, i.e., rules, their guards, and their subconfiguration. During the execution of a program run–time objects are created by instantiating static objects. In PJ, for instance, run–time objects are created when a subconfiguration is instantiated after the selection of a rule. The creation resp. instantiation requires the synchronization of the interpreter and animator for keeping their private data structures consistent. Figure 6 gives an overview of an PJ interpreter–animator synchronization with respect to the execution and animation steps given in the Subsections 3.2 and 3.3.

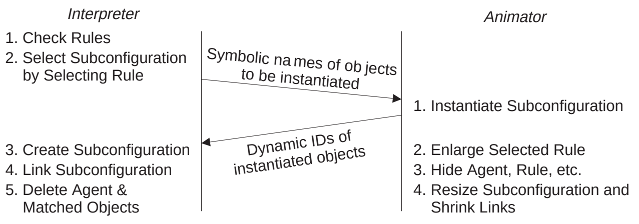
Interpreter
Animator
1. Check Rules
2. Select Subconfiguration by Selecting Rule
Symbolic names of objects to be instantiated
1. Instantiate Subconfiguration
Dynamic IDs of instantiated objects
3. Create Subconfiguration
4. Link Subconfiguration
5. Delete Agent & Matched Objects
2. Enlarge Selected Rule
3. Hide Agent, Rule, etc.
4. Resize Subconfiguration and Shrink Links

Figure 6. PJ Interpreter–Animator Interaction

When starting the animation the data structure of the interpreter has to be configured with the program. Whenever the graphics (program) is being modified during the execution by the editor the interpreter is notified in order to update its logical data.

The temporal animator–interpreter synchronization is realized by commit signals for the end of each animation operation. The interpreter invokes an animation operation by a unique operation identifier. The animator notifies its completion by sending the corresponding commit signal.

5.2.2. Interface

Different applications require different levels of automation in the animation control. For some applications, e.g., when implementing a debugger, it is necessary to manipulate the fine–grain structure of a program where other applications only require a basic synchronization during the execution at particular points.

We have defined an animation interface supporting different abstraction levels with respect to different levels of automation. Table 1 gives an overview of the five abstraction levels and relates it to the classification of 3–D animation and their techniques given in Section 4. With increasing abstraction more automation with respect to the execution is incorporated in the animator. That means, that at the lower levels the individual graphical objects can be controlled where at higher levels the individual behavior, e.g., pattern matching, is performed by the animator or by the graphical objects, respectively. Level 1 and Level 2 completely depend on the individual primitives and objects of the visual representation. The higher levels should be independent to that animated representation. Level 1 and Level 2 of the following interface are thus introduced by the example of PJ primitives and objects. The individual levels are defined as follows.

Level	*Technique*	*Interface*
2	Parametric Keyframe Animation	Primitive Level
3	Algorithmic Animation	Object Level
3	Task level Animation	Execution Level
4	Actor–based Animation	Trigger Level
4	Actor–based Animation	Automatic Level

Table 1. Definition of Levels

Primitive Level. This level provides the manipulation of basic drawing primitives. For PJ that means the control of the transformations of closed and open contours. Transformations of the primitives are performed by assigning new values to parameters such as size, scale, position, or color. The individual parameters are changed to their new value by a frame–wise interpolation (*parametric keyframe animation*).

Object Level. This level gives control to the basic syntactical objects which are composed of basic drawing primitives. For PJ basic objects are, ports, constants, rules, links, channels, etc. The manipulation of objects requires dedicated algorithms for the manipulation of their subelements and is thus comparable to *algorithmic animation*. Note that this level completely depends on the syntax and semantics of the underlying visual mean.

Execution Level. This level permits the manipulation of the individual actors of a program, e.g., agents and functions. The evaluation of conditions is performed by the interface operations. Thus, significant behavior of the actors still has to be controlled by an interpreter. For PJ we can distinguish three operations with respect to the corresponding execution model: *select_rule*, *solve_agent*, and *recompute_layout*. This level depends on the individual tasks performed through the execution and can thus be denoted as *task level animation*.

Trigger Level. This level provides operations to run the animation in synchronous mode by individually triggering the execution of each actor. The actors can be automatically triggered by a global strobe or the user may individually choose the order of execution. The behavior of each actor, e.g., pattern matching, is performed by the animator.

Automatic level. Almost the complete execution is asynchronously computed by the animator. The animator is given a subset of the program in form of a set of actors — including the actors being generated through the execution — and executes them until the program deadlocks or terminates.

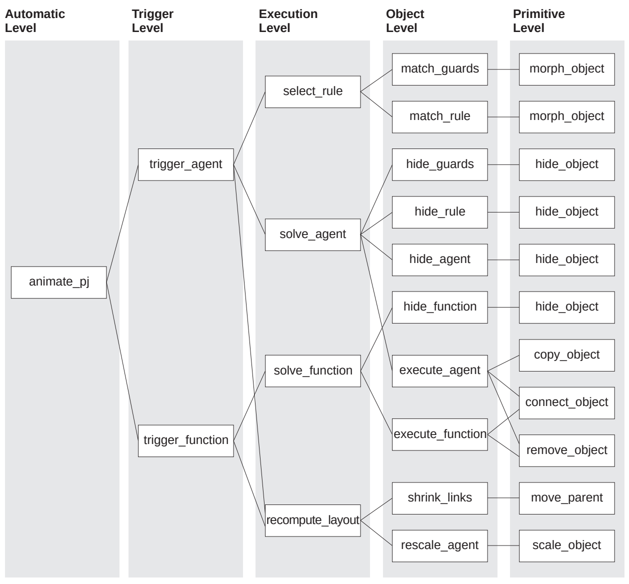

Figure 7. Animation Operations

Figure 7 gives a listing of the relevant operations by the means of their hierarchical relationships. An operation at level i typically implements a higher–level execution based on a set of level $i-1$ operations.

The interface additionally provides operations for the **animation control**. These are operations to start, pause, continue, and step through the animation. Further operations provide a redraw, sound control, and the monitoring of active areas.

6. Implementation

We have proved our concepts by the implementation of the distributed debugging environment for Pictorial Janus, namely JIM (Janus In Motion). The implementation covers a PJ editor/animator and an interpreter with debugging facilities. The current animator supports the animation control by a set of parallel interpreters though the debugger is implemented as one sequential process. For the animation interface we have implemented primitive and object level operations as well as most parts of the animation control. The editor/animator is implemented by the use of the X11/Windows editor toolkit EOS (Editor Object System) [11]. For the interprocess communication we use PVM (Parallel Virtual Machine) [1] which supports over 30 platforms. JIM is implemented in C++ under SunOS 4.X and has been ported to Solaris, IRIX, and WindowsNT.

JIM additionally incorporates external PJ functions so that JIM can be used as an open system integration platform. The simple concept of external functions allows to control arbitrary software and hardware components. Technically, an external PJ function spawns an additional process. The values of the PJ input messages are send to the spawned process. The received messages are displayed as output messages of the external function in JIM. Figure 8 gives an example for coupling two VHDL Synopsys (VSS) simulators. The simulators are integrated via the Synopsys C language interface (CLI). One simulator simulates the behavioral VHDL model of the DLX RISC CPU [2]. The other simulator simulates the connected main memory. The exchanged messages, i.e., read/write requests and instructions, are visualized as PJ messages. Figure 8 gives a screenshot of the animation of that system. The main windows of the spawned simulators are shown in the background left and right of the JIM main window. A waveform display gives the history of the ports. Since the exchanged messages can be easily modified during the animation JIM serves well as a testbench for VHDL models, e.g., for evaluating the CPU's instruction set. Note, that visualization and animation also applies delta–cycle simulations when VSS generates no waveform trace.

7. Conclusion

In this article we have introduced a new approach to the distributed implementation of an interactive real–time 2–D animation and design environment for visual programming languages. We have introduced the individual components of such a system and outlined their interaction emphasizing the interpreter–animator interaction. The system has a flexible animation interface supporting multiple levels of automation. Although we have proved the concepts by the implementation of a design environment for Pictorial Janus, the generic concepts of the architecture should be well applicable for other visual languages following the principles of concurrent agents communicating by exchanging messages. Additionally, the decoupling from the interpreter makes the implemented animation front–end applicable as a visualization tool of other execution models. We are currently thinking about the replacement of the PJ interpreter by a Petri–Net simulator as well as the integration of event–driven hardware simulator for the IEEE standard hardware description language VHDL'87. Our future investigations also include the application of complete visual languages for manufacturing process monitor and control.

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 support.

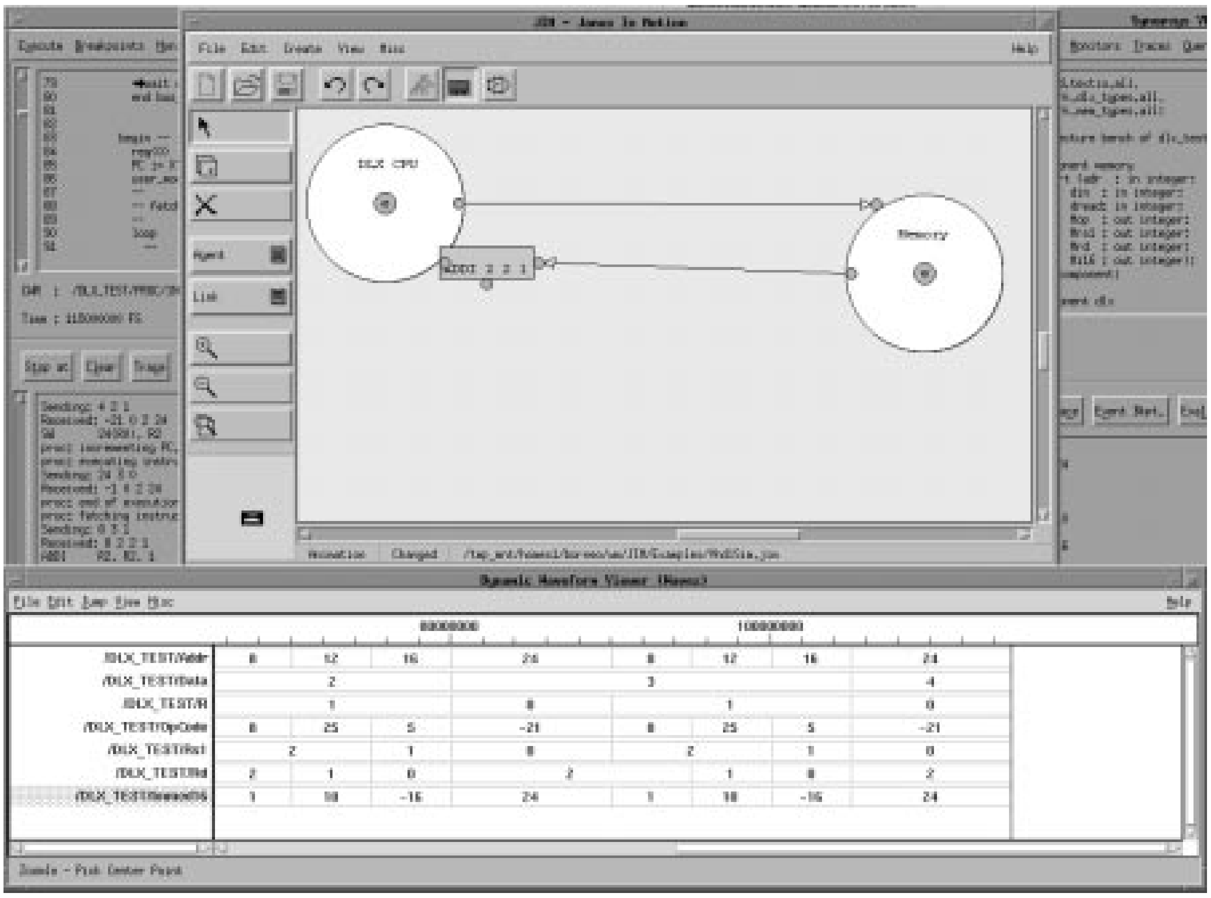

Figure 8. JIM as Simulation Backplane coupling two Synopsys (TM) Simulators

References

[1] A. Beguelin, et al. PVM and HENCE: Tools for heterogeneous network computing. In J.J. Dongarra and B.Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*. North-Holland, Amsterdam, 1993.

[2] P.J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1996.

[3] M.H. Brown. *Algorithm Animation*. MIT Press, 1987.

[4] P. Carlson and M. M. Burnett. A seamless integration of algorithm animation into a visual programming language with one-way constraints. In *Proceedings of the International Workshop on Constraints for Graphics and Visualization, Cassis, France*, September 1995.

[5] W. Citrin, D. Doherty, and B. Zorn. The design of a completely visual oop language. In M. Burnett, A. Goldberg, and T. Lewis, editors, *Visual Object-Oriented Programming*. Prentice-Hall, 1995.

[6] W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *Proceedings of the IEEE Workshop on Visual Languages*. IEEE, September 1995.

[7] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Eighth European Conference on Object-Oriented Programming*, Berlin, 1994. Springer-Verlag.

[8] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[9] K. Kahn and V.A. Saraswat. Complete visualizations of concurrent programs and their executions. In *1990 IEEE Workshop on Visual Languages*, Oct. 1990.

[10] H.-J. Kaufmann. *EDIS: Eine objekt–orientierte Software–Architektur fuer graphische Editoren*. PhD thesis, Paderborn University, 1994.

[11] H.-J. Kaufmann, Th. Kern, and Rui Zhao. *Detailed Functional Specification EOS 1.0 – Technical Report BT-HCI 94*. Cadlab, Paderborn, Germany, 1994.

[12] E. Kraemer and J.T. Stasko. The visualization of parallel systems: An overview. In *Journal of Parallel and Distributed Computing*, volume 18. Academic Press Inc., Dordrecht, Netherlands, 1993.

[13] Nadia Magnenat-Thalmann and Daniel Thalmann. *Computer Animation*. Springer, Wien, 1985.

[14] Nadia Magnenat-Thalmann and Daniel Thalmann. *Synthetic Actors in Computer-Generated 3-D Animation*. Springer, Tokyo, 1987.

[15] D.L. Miller-Karlow and E.J. Golin. vVHDL: A visual hardware description language. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*. IEEE CS Press, 1992.

[16] A. Repenning and T. Summer. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3), March 1995.

[17] V.A. Saraswat, K.M. Kahn, and J. Levy. Janus-A step forward towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*. MIT Press, Oct. 1990.

[18] J. T. Stasko. The path-transition paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3), September 1990.

[19] S. L. Tanimoto. Towards a theory of progressive operators for life visual programming environments. In *Proc. 1990 IEEE Workshop Visual Languages*, pages 80–85, Skokie, Illinois, October 1990.