Complete Visual Specification and Animation of Protocols*

Wolfgang Mueller

Cadlab
33102 Paderborn, Germany
e-mail: wolfgang@cadlab.de

Georg Lehrenfeld, Christoph Tahedl

Heinz Nixdorf Institut
33098 Paderborn, Germany
e-mail: {georg,tahedl}@uni-paderborn.de

Abstract— This article discusses the application of Pictorial Janus (PJ) for the rapid development and analysis of protocols by animation and complete visualization. In order to make PJ applicable in the context of hardware description we first extend PJ by timing facilities (Timed PJ) and introduce an approach for integrating VHDL models into this visual framework preserving the simulation semantics of VHDL. We finally give the example of the specification and animation of a non interlocked protocol.

I. Introduction

The specification of protocols is a key activity in parallel systems design. Timing as well as functional properties, such as deadlocks, duplicated or lost packets, are critical issues in protocol design and specification. Even if the utmost care is exercised during protocol design there is a strong possibility for not catching all errors. Thus, protocol verification and protocol validation at each level of abstraction is of utmost importance[7]. The visual runtime inspection during the simulation can be improved by an advanced animation. Visualization and animation tools supporting the analysis and execution of concurrent programs are expected to play an important role in the next years [11, 6]. However, only little attention has been paid for the animation in electronic design automation. Most of the known design environments provide a statical representation of the dynamics of programs. Only a few provide basic animation by blinking or highligthing the current state or statement.

Pictorial Janus (PJ), introduced by Kahn and Saraswat in [9], is the complete visual representation of the concurrent logical programming language Janus. PJ was the first representative of a programming language with a complete visual representation providing a smooth inherent animation of the program's execution. In this article, we extend PJ for timing specification (Timed PJ). We think that Timed PJ, due to its complete visual representation, provides new directions in the specification and analysis of protocols. In order to make use of Timed PJ as a visual animation and high level specification front–end for VHDL we introduce the integration of the VHDL simulation cycle with the execution model of Timed PJ (Pictorial (V)HDL).

The remainder of this article is organized as follows. We first discuss related work in visual representation and animation. Section II introduces PJ and extends the concepts of PJ by the notion of time (Timed PJ). Section III introduces Pictorial (V)HDL. Section IV discusses the application of Timed PJ for the specification and animation of bus protocols. The article closes with a conclusion and outlook.

II. Related Work

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 (Program–State Machines) [8]. A recent trend shows the use of advanced graphical environments for VHDL designs, such as Process Model Graphs (PMG)[1], SpecCharts[8], Speedcharts[1], and VisualHDL[2]. All of these means mainly provide the graphical representation of the control flow–e.g., hierarchical states annotated by VHDL code, state transitions labeled by conditions, processes connected by communication links–by the means of circles, (embedded) boxes, and links between them. Those means which are intended to aid VHDL modeling support the entry of process statements and expressions in textual form. vVHDL additionally gives an iconic representation of the various sequential statements [13].

Some of these systems provide very basic animation facilities by blinking or changing the color of the currently active state symbol or statement. For the advanced animation of concurrent systems various frameworks are available. An overview can be found in [11]. Most systems incorporate visualization tools for the generation of graphical representations, such as call graphs, execution history, temporal orderings, traces, performance diagrams, XY plots, in order to achieve an abstraction the animation is based on. In this context we distinguish performance and execution animation. In execution animation, most

*Published in: Proceedings of the IFIP Conference on Hardware Description Languages and their Applications, August 30 - September 1, 1995, Tokyo, Japan

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

systems permit to change the graphical context of active nodes and/or provide the motion of objects by switching and moving icons (e.g., HENCE[2]). In most systems the user is required to modify the original program coding and integrating the animation scenes by hand (e.g., BALSA[5]).

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 drawing or retrieving the data dependencies from the textual program code. In contrast to the above animation frameworks, 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. Additionly, PJ provides a smooth continuous animation of its execution providing continuous movements and actions–not just blinking objects or changing the color. Nevertheless, PJ lacks the notion of time and no investigation for the foreign code integration has been done so far.

III. Pictorial Janus (PJ)

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

A. PJ Objects and their Topological Semantics

A PJ program is composed of graphical primitives, i.e., closed contours and connections between them. The meaning of a closed contour is independent from its geometrical representation and graphical context, i.e., shape, size, color, etc. The meaning of a PJ object is only given by the topological relationships touching/not-touching and inside/outside to other PJ primitives. Basically, PJ distinguishes messages, agents, functions, and connections. Each of these elements may have ports in order to establish a connection to other objects. We distinguish external and internal ports corresponding to their topology, i.e., whether the port is attached from the inside or to the outside of an object. Figure 1 gives a list of the different PJ elements. In that figure ports are filled grey in order to emphasize their contours.

Constants and lists are referred to as messages. Constants hold values. Lists are used to set up more complex data structures carrying messages between agents and functions. Different elements may be connected by links, which are represented by undirected lines. In most cases a link is used to connect an internal port with an external one. E.g., the internal port of a list element or constant may be linked to the external port of another list element. Links represent data dependencies.

Functions and agents consume and produce messages. An agent is given by a closed contour with a set of external

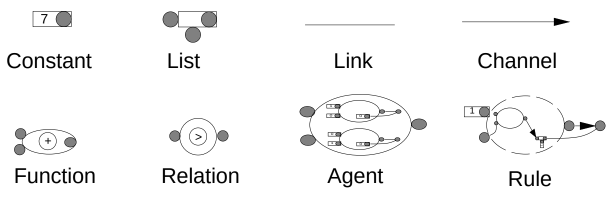


Fig. 1. PJ Objects

(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 body (contour and ports). The rule defines the behavior of an agent with respect to different input patterns (guards). The guards are defined outside the rule's contour whereas the behavior (subconfiguration) is defined inside the rule's contour. Within the guards, the relation objects are used to define constraints between messages. The new objects (subconfiguration), i.e., messages, functions, and agents, which are being created when matching the individual rule, are located inside the rule's contour. Instead of explicitly specifying the behavior of an agent a call arrow may instantiate the rules from a another agent. This is similar to a procedure call in imperative programming languages. Call arrows give the control flow in a PJ program. Functions have a predefined behavior denoted by a symbol inside their contours. So–called channels establish directed connections between two external ports. Their intuitive meaning is to "send" messages to other agents. For a more compact representation of PJ drawings two shorthands are introduced in Figure 2.

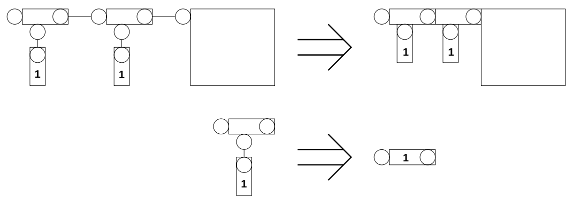


Fig. 2. Shorthands

We briefly outline PJ by the example of an elevator controller. Figure 3 shows the corresponding agents with two ports for holding the current and the requested floor. The current floor is given by the value 3. At the other port the values 4 and 1 are scheduled. The behavior is specified in form of 3 rules. The first one decrements (−−) the current floor when the current floor is greater than the first message, i.e., the first element of the requested floors. In the third rule the current floor is incremented. In both cases a new subconfiguration with a function incrementing/decrementing the current floor and the recur-

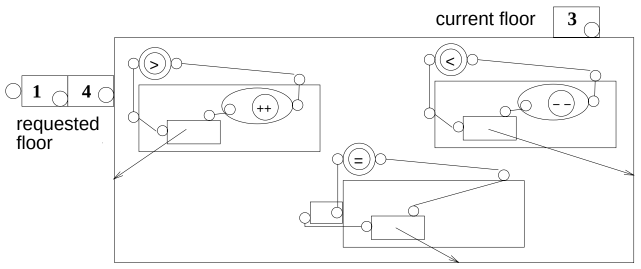


Fig. 3. Elevator Controller

sive replacement of the agent is spawned. The last rule matches if the requested floor equals the current floor. In that case the current floor is simply saved and the second message of the queued requests becomes the new request.

B. Execution

PJ defines a set of concurrently communicating agents sending and receiving messages. In the case that an event has been detected at any external port of an agent, i.e., any message is scheduled, the agent resumes and executes the behavior of one rule which has successfully matched, i.e., the corresponding subconfiguration is created and linked to the existing configuration. The rule is being selected by a pattern matching of the incoming messages w.r.t. the guards of each rule. After the selection the matched objects and the original agent is being destroyed. A function executes accordingly when all messages at its input are available.

In order to introduce the notion of time to PJ for real–time applications we introduce the concept of timers[3]. 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;
    od;
od;
```

Timers execute on the advance of the current time. The above algorithm advances the time if no other events have occurred at any agent or function. In that case the current time is updated to the time the next timer is expected to

[3]The introduced concept of time is comparable to that of the CCITT standard specification language SDL[12].

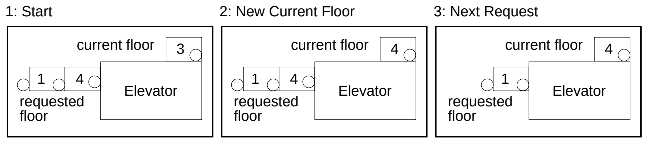


Fig. 4. System Level Animation

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.

C. Animation

A PJ drawing is a snapshot of a computation. Moreover, PJ provides the complete visualization of the computation by an inherent advanced animation. Due to their rotational and dilational invariance PJ objects may vary in size and outlook during the execution or animation, respectively. When the animation advances valid PJ follow–up configurations are derived from the current configuration. We distinguish the animation on *system level* and on *component level.*

The *system level animation* displays the agents as black boxes. Only the agents and the scheduling and consuming of messages is displayed at this level. Figure 4 gives three snapshots when animating the example in Figure 3 at system level. Frame 1 shows the start configuration. In Frame 2, the current floor is incremented when matching the upper left rule. Frame 3 presents the configuration when the current floor equals the request and the first message has been removed from the input queue.

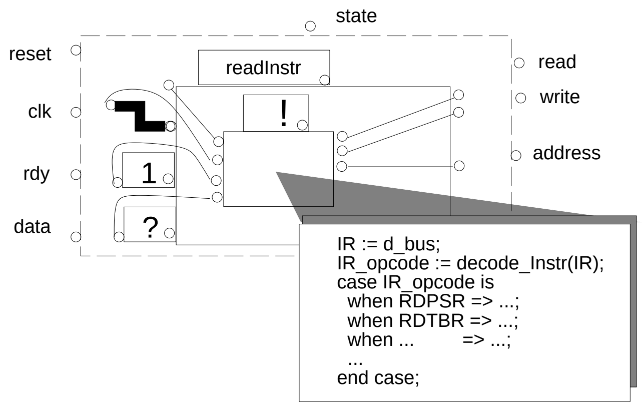
The *component level animation* completely animates the pattern matching, the selection, creation, and linking of a subconfiguration. Once a rule has been selected the other rules disappear and the selected rule is continuously enlarged until the guard and its contour overlap the corresponding objects of the agent. Thereafter, the matched objects including the guard as well as the agent's and the rule's contour smoothly disappear animating the linking of the subconfiguration. Then, the shrinking of links animates the movement of messages. Due to limitations in space, we have to restrict our view to the system level animation in the remainder of this article.

IV. Pictorial (V)HDL

In order to make use of PJ to capture and animate hardware descriptions we have to investigate the integration of VHDL into Timed PJ's visual framework by introducing Pictorial (V)HDL (PHDL). The integration is required to make use of existing VHDL code. Additionally, the Timed PJ/VHDL co–specification overcomes some disadvantages of Timed PJ (see also [12]). The integration has two aspects. Firstly, the interfacing of VHDL with PJ drawings has to be defined. Secondly, both execution models have to be unified preserving the execution semantics of VHDL without conflicting the animation semantics of Timed PJ.

A. The PHDL Capture

In order to assign VHDL programs to PJ rules we introduce the concept of foreign PJ rules. A foreign rule is associated with a set of (sequential) VHDL statements. This introduces a concept for assigning VHDL processes to PJ rules. There is an implicit wait statement of the form "**wait until** guard;" assumed as the last statement where guard is the guard of the enclosing PJ rule. Each time a foreign rule is being matched the VHDL statements are executed in sequential order suspending after the last statement[4]. The VHDL port signals are associated with the PJ ports by a port map declaration[5]. In PHDL, the linked output message waiting for a driving value is denoted by a **!**–symbol when associated to an outport. The symbol is replaced by the driving value of the associated signal as soon as it changes its value. Correspondingly, the **?**–symbol is used in a guard if the value of a message has to be passed to an associated VHDL port The value is assigned as soon as the rule matches. Additionally, that signal is set active in order to invoke the propagation of the effective value to the net of embedded VHDL objects.



```
IR := d_bus;
IR_opcode := decode_Instr(IR);
case IR_opcode is
  when RDPSR => ...;
  when RDTBR => ...;
  when ...        => ...;
  ...
end case;
```

Fig. 5. Example for Integrating VHDL into Timed PJ

We outline the application of PHDL by the example given in Figure 5. This example defines the decoding of the instruction of a simplified microprocessor. The rule matches if the microprocessor is in state *readInstr*, there is a down edge at the clock input[6], the ready signal applies, and there is a message available at the data input. Inside the rule's contour the contour of the agent which embeds the VHDL code which is associated with the environment by the port map (d_bus => data, a_bus => address, state => state, ...); The constant at the upper top of the agent denotes the next state of the microprocessor. In our example, the next state is assigned from the embedded VHDL process w.r.t. the decoded instruction, i.e., whether there is a read request, a write request, etc.

[4]This requires the concept of persistent PHDL agents.

[5]This requires the assignment of object identifiers to PJ ports.

[6]The clock message is shaped corresponding to its semantics CLK–DOWN in this example.

B. The PHDL Execution Cycle

In this subsection, we sketch the integrated Timed PJ/VHDL simulator concentrating on the core concepts of the execution rather than discussion details, such as shared variables and postponed processes. This is mainly due to the complexity of the full VHDL'93 simulator and the richness of the VHDL'93 language. A formal specification of the full VHDL'93 simulator can be found in [3]. The following algorithm sketches the very basic steps of a VHDL simulator.

```
While current time < TIME'HIGH do
  if no signal events and no expired timeouts at $T_c$ then
    update current time $T_c$;
    compute new signal events and new timeouts;
  fi;
  check for (events and true condition) and expired timeouts;
  execute processes;
  wait until all processes suspended;
od;
```

Until the current simulation time is advanced to the final simulation time (TIME'HIGH) each simulation cycles performs as follows. If any event on signals is detected or any timeout expires (wait for 0 ns) as a result of the previous simulation cycle a delta cycle has to be performed, i.e., the current simulation time is not advanced. Processes which are suspended on these events or on expired timeouts have to be resumed. If a process has been suspended on a condition the condition has to be checked. Thereafter, all resumed processes are executed until each process is suspended again. Processes may generate events for the current simulation time when being executed. If no event occurs on any signal or any timeout expires at the current time T_c the current simulation time is updated to the time when the next event occurs. Thereafter, these events are generated and the expiration of all timeouts expiring at the new current time (t_n) are computed. As a result processes are resumed and executed again, etc.

In order to extend the above VHDL simulator by the execution model of Timed PJ as it is defined in the previous section we have to extend the notion of a VHDL event. In PHDL, we interpret event as: (1) event on any external port of a PJ agent when any message is scheduled, (2) expiration of any PJ timer, (3) event on signals w.r.t. the VHDL'93 LRM [14], or (4) expiration of any VHDL timeout generated by a wait for statement. Under these assumption the PHDL execution cycle is sketched by the following steps.

```
While current time < TIME'HIGH do
  if no events at current time then
    update current time;
    compute new events;
  fi;
  execute processes/agents/functions
                sensitive on events and/or true condition;
  wait until all processes/agents/functions are suspended;
od;
```

If any of the previously defined events have occurred it has to be checked whether any VHDL process/Timed PJ function may resume or any rule of a Timed PJ agent [7] may match the newly scheduled messages. If the check evaluates to true the corresponding process/function/agent is executed until it suspends. Timed PJ functions are defined to be suspended on their destruction. Recall here, that each function is destroyed when it has produced its output. A Timed PJ agent is defined to be suspended after it has executed the matched rule. In the case that no events have been generated for the current simulation cycle the current time is advanced to the occurrence of the next event. These events may be a signal event, an expired timeout, or an expired PJ timer.

Comparing the above PHDL execution cycle with the previously sketched VHDL execution cycle it is easy to see that the PHDL simulator supersedes the behavior of the VHDL simulator and thus is able to run pure VHDL models as defined by the IEEE standard.

V. Example

For our final example we have chosen a non interlocked bus protocol (see [15]) since this protocol is safety critical and thus serves best for demonstrating problems during the analysis. The protocol specifies the data transfer from a source to a destination. The transfer is synchronized by a data request (DR) and a data accepted (DA) line.

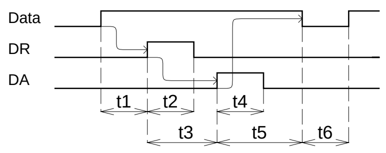


Fig. 6. Non Interlocked Protocol Timing

The timing diagram in Figure 6 depicts the behavior of that protocol. The source first puts data on the bus. After a delay t_1 it drives DR to high. The destination is sensitive to DR and sets DA. The source which is sensitive to DA removes the data from the bus. When modeling signal changes in the timing diagram by transactions (generated by signal assignments) results in the following VHDL specification.

[7]This covers guarded processes which are integrated as foreign code.

```
source:                               destination:
  process begin                         process begin
    Data <= Buffer after t6;              wait until DR = '1';
    DR <= '1' after t1;                   Buffer <= Data;
    DR <= '0' after t2;                   DA <= '1' after t3;
    wait until DA = '1';                  DA <= '0' after t4;
    Data <= null after t5;              end process;
  end process;
```

When simulating the above VHDL program the case $t_4 > t_1 + t_3 + t_5 + t_6$, results in the timing given in Figure 7. This timing is due to the preemptive scheduling of waveform elements performed by a VHDL signal assignment statement. As even shown by this simple example, VHDL programs are sometimes cumbersome in the debugging of timing failures. Considering the VHDL output it is not obvious that the value of t_4 results in the unexpected behavior and that the correct timing can be achieved by decreasing t_4 or increasing t_1, t_3, t_5, or t_6. In that case the user is confronted with the detailed analysis of the signal attributes. To our knowledge, the analysis of attributes during the multiple simulation cycles is not well supported by any of the existing simulation environments. These situations require dedicated tools, such as timing diagram editors generating VHDL code[4, 10]. Nevertheless, since they do not support backannotation their application seems to be limited to the initial timing specification.

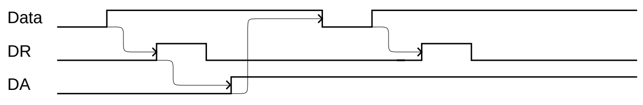


Fig. 7. VHDL Simulation Timing Diagram

Since there is no preemptive scheduling in Timed PJ values do not necessarily have to be reset in Timed PJ specifications. The reset is implicitly given when "consuming" the message. Timers can be used to model timing relationships between events where the arrival of a message at a port refers to an event in the timing diagram, e.g., DR high and DA high. Thus, a timer models the distance between two events.

For additional timing intervals in the following Timed PJ example, we set $t_8 = t_1 + t_3 + t_5$, $t_9 = t_8 + t_6$, $t_{10} = t_1 + t_3$, and $t_{11} = t_5 + t_6$. t_1–t_{11} define the set of the relevant intervals required for an adequate analysis of the timing constraints. Figure 8 and 9 show the PJ specification at component level w.r.t. the timing diagram given in Figure 6[8].

In Figure 8, the rule of the source agent sends the data and the '1'–message via Data and DR. The rule is applied if a '1'–message is matched at the DA input. Figure 9 defines 2 rules. If the destination matches the data message and a '1'–value at the DR input it notifies the receipt

[8]Note, that for a more detailed analysis message can also be used to represent edges indicating a signal change.

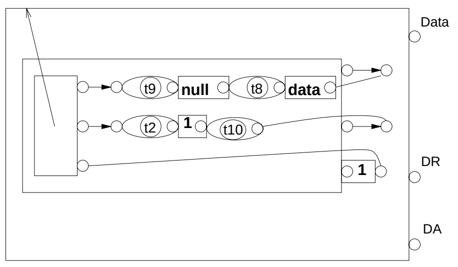


Fig. 8. Component Level Specification of the Source Agent

by a '1'–message at the DA output. The second rule of this agent just removes the null–value from the Data input. In order to represent timing constraints, messages are headed by timers which postpone the delivery of the message by a time t_i. This specifies the timing distance to the generated event where the appended timer models the duration the signal is assumed to be high. Consider, for instance, the upper rule of the destination agent. This rule schedules two timers (t_{11} and t_4) which enclose a '1'–value at the DA channel. t_{11} models the distance between two DA–High events where the timer t_4 models the duration of the DA–High signal.

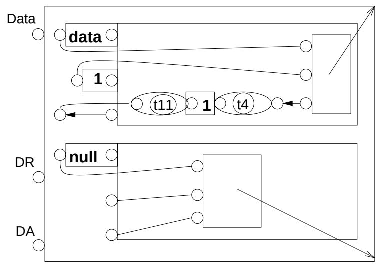


Fig. 9. Component Level Specification of the Destination

Figure 10 gives 6 animation snapshots being produced when animating the configuration based on the definitions given in Figure 8 and 9. In order to concentrate on the communication aspects of the protocol we outline the example by running a system level animation. This Timed PJ configuration was specified in order to run an iterative animation, i.e., the last configuration of a transfer cycle (Frame 6) is the predecessor of the first configuration of the next cycle (Frame 2). Thus, an arbitrary number of animation cycles can be performed if required. In the given example, Frame 2 shows the messages and timer being created when matching the rule of the source agent. The next configuration results from the expiration of t_{10}. On the receipt of the data message and the expiration of timer t_{10} the destination agent applies its first rule creating a message as well as t_4 and t_{11} on DR. The destination agent matches the values at the Data and DR input by applying its first rule which leads to the configuration in Frame 3. After the expiration of t_8 and t_4 the destination matches and removes the null–value at the data input. Next, t_{11} expires and applies a message to the DA input of the source (Frame 5). The source agent immediately matches the DA–event and schedules further values and timers to Data and DR. The follow–up configuration after the expiration of t_9 in shown in Frame 2. The remaining animation iterates between the configuration given in Frame 2 and the configuration given in Frame 6.

The animation, as depicted in Figure 10, allows the detection of timing anomalies. Since each message is headed by a timer the value of the message is not directly scheduled to the port but it is scheduled after the expiration of the timer. This models the delayed application of values. A timing error occurs if at any input two values are scheduled with no timer between them. In the context of digital components this means that a connection is driven by two values at the same time which refers to an error in protocol design, i.e., to a functional or timing error. Consider, for instance, Frame 6 in the above example. Under some invalid timing conditions, timer t_8 may expire before the expiration of the first t_9.

Another timing error occurs if a new value is scheduled when the value on that channel is still valid. That means, if two timers are scheduled with no message between them. This can be the case if DA is not reset before the start of the next transfer cycle (see also Figure 7). In that case, t_4 will be kept through the Frames 4-6 of the first cycle. If t_4 still exists in the second cycle until Frame 3, timer t_{11} is scheduled directly after t_4 which marks a violation of the protocol.

The previous example demonstrates that in Timed PJ, functional errors and anomalies can not only be detected by a visual inspection of the static drawing. Additionally, an advanced visual inspection can be performed by animating the specification. The iterative animation easily allows to detect functional errors. Lost messages and deadlocks will stop the animation after a few cycles. Wrongly duplicated messages overflow the input queue at an external port. Wrongly linked objects result in a visual chaos. In almost all cases it is easy to detect the intermediate configuration which is responsible for the chaotic behavior when starting the initial configuration for a second run or stepping back to previous configurations. A visual inspection of the individual intermediate configuration allows us to detect the functional error at any step of the animation. Since the scheduled values are kept although the timing constraints are hidden the future configurations can be investigated in details by continuing the animation. This allows, for instance, to analyze whether a invalid local timing influences the further general behavior and results in the instability of the total system. In Timed PJ, the bug is really visible and can be corrected or removed by modifying the drawing. Compared to a VHDL simulation this is a qualitative improvement

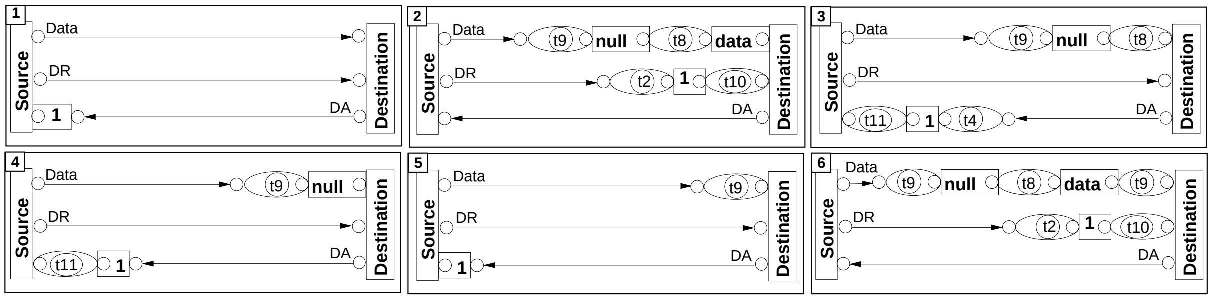


Fig. 10. Animation of Non Interlocked Protocol

when analyzing the behavior during the very first design phases.

VI. Conclusion and Outlook

We have demonstrated that Pictorial (V)HDL provides additional qualitative means for the prototyping, specification, and debugging of highly concurrent systems. Additionally, we have shown how VHDL designs can be smoothly integrated into the visual environment of Timed PJ preserving the simulation semantics of VHDL. Due to limitations in space we have only sketched the integrated PJ/VHDL execution cycle. Based on the work introduced in [3] a complete formal model will be available soon. In this paper, we have demonstrated our approach by the example of a low level bus protocol. Additional investigations show the specification of high level transport protocols, such as the Ethernet CSMA/CD protocol, as well as the transformation of general SDL specifications to Timed PJ [12].

Nevertheless, Pictorial (V)HDL has not been introduced to replace the existing analysis tools, like simulators or protocol verifiers, nor to replace existing means, like timing diagrams, hierarchical state machines, control flow graphs, etc. Our purpose when presenting this language is to indicate new directions in the visualization and advanced animation which still have to find a place in general systems design. Nevertheless, we see some problems for getting a Pictorial (V)HDL–like language accepted by the general hardware designer. This mainly refers to an educational problem since electrical engineers as well as most computer scientists are not well educated in concurrent logic programming languages.

We have implemented a basic Timed PJ Editor under X11R5 and MS-Windows based on our in–house editor toolkit EOS. Significant effort went into the implementation of the animation and debugging facilities which has recently been completed for SunOS. After a final tuning we expect to start the VHDL integration soon.

Acknowledgements

We acknowledge the comments of F.J. Rammig and U. Hudson when reviewing this work. We would like to thank Marita Duecker and Bettina Hoischen for implementing parts of the PJ environment. Special thanks are dedicated to H.J. Kaufmann who greatly helped us through the EOS toolkit.

References

[1] J.R. Armstrong and A. Honcharik. Rapid development and testing of behavioral models. In J.P. Mermet, editor, *Fundamentals and Standards in Hardware Description Languages.* Kluwer Academic Publishers, Dordrecht, Netherlands, 1993.

[2] A. Beguelin, J.J. Dongarra, G. Geist, and R. Manchek. Graphical development tools for network–based concurrent supercomputing. In *Proceedings of Supercomputing'91*, 1991.

[3] E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by EA–Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics For VHDL.* Kluwer, Boston/London/Dordrecht, 1995.

[4] G. Borriello. Formalized timing diagrams. In *Proceedings of the 1992 Eurpean Conference on Design Automation*, 1992.

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

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

[7] D.L. Dill, A.J. Drexler, A.J. Ilu, and C.I. Yang. Protocol verification as a hardware design aid. In *Proceedings of the ICCD'92.* ACM Press, 1992.

[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] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, and A.Silburt. Intergrating behavior and timing in executable specifications. In *Conference proceedings of the CHDL'93.* OCRI Publications, 1993.

[11] 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.

[12] G. Lehrenfeld, W. Mueller, and C. Tahedl. Transforming SDL into a complete visual representation. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, 1994. (In press).

[13] 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.

[14] The Institute of Electrical and Electronics Engineers, New York, NY, USA. *IEEE Standard VHDL Language Reference Manual-IEEE Std 1076-1993*, 1994.

[15] A.J. van de Goor. *Computer Architecture and Design.* Addison–Wesley, 1989.