Beyond VHDL: Textual Formalisms, Visual Techniques, or Both?

Franz J. Rammig
Heinz Nixdorf Institut, Universität-GH, Paderborn
franz@uni-paderborn.de

Abstract

Since a couple of years VHDL is the dominating Hardware Description Language. There are very good reasons for this and simply the existence of VHDL as standardized language had a major impact on the advance of high level design techniques. In this paper some ideas about specification and modelling techniques beyond VHDL curently carried out in the author's research group are discussed. One approach is to integrate formal specification techniques like Evolving Algebras (EA) and Z into a VHDL-oriented design environment . Other approaches concentrate on the potential of visual modelling techniques. Here techniques based on Parallel Logic Programming like Pictorial Janus (PJ) or such ones based on higher order Petri Nets are under investigation.

1. Introduction

VHDL may be seen as a formal modelling tool. From the point of view of more abstract systems designers, VHDL is a modelling language and not at all a specification means. The analogy to software engineering makes this statement evident. No software engineer would treat a program in C++, say, as a specification. In this part of the paper two techniques of more abstract specifications are discussed shortly: Evolving Algebras (EA) and Z. Both have connections to VHDL. One of the existing definitions of a formal semantics of VHDL has been done in EA [BGM95] while Z is studied by the SDDL-group of DaSC [Bar95].

1.1. Evolving Algebras

EAs have initially been defined by Gurevich [Gur93]. In this paper an introduction very similar to this one in [BGM95] is used. EAs can be understood as "pseudocode over abstract data", without any particular theoretical prerequisites. The abstract data itself is given by elements of sets (domains and unverses) denoted by capitalized words. In abstract specifications these sets need not to be specified further. The operations allowed on universes are represented by partial functions. Relations are denoted by their associated boolean-valued membership function. So heterogeneous structures $(D_1, ..., D_n;\ f_1, ...f_m)$ with the domains D_i and functions f_j have to be considered. Such structures without relations traditionally are called *algebras*. In the approach of EAs such algebras (called *state algebras*) are used to desribe states of the system to be specified. EAs are operational by nature, i.e. systems are specified by *states* and *state transitions*. *State transitions* are represented as transformations of static algebras. Such changes are obtained by executing *update instructions* of form $f(t_1, ..., t_n) := t$.

This can be read as setting the value of function f at the given arguments to the value t. Obviously 0-ary functions play the role of *variables* in imperative programming languages and the above update instruction becomes a simple value assignment in this case. As partial functions are considered the special value *"undef"* is possible. Now these update rules need to be guided in some way. For this purpose a *sequential EA-Machine* is defined as a finite set of transition rules of form **if** *Cond* **then** *Updates*.

Here *Cond* (called condition or guard) is a first-order logical expression. This expression becoming true triggers the simultaneous execution of all update instructions in the finite set *Updates*. The following simple example, taken from [BGM95], illustrates sequential EA-Machines:

if *Condition*
then *A := B*
B := A

This little example defines the simultaneous update of the 0-ary functions *A* and *B*. Since these two updates are performed strictly in parallel the two values of *A* and *B* simply become exchanged. This swap of values is performed whenever *Condition* evaluates to *true*. Additional parallelism is introduced by allowing formal variables in the update instructions where the domain of such formal variables is declared in the guard of the transition rule. In the following example, simultaneously the first element of all lists within a certain specification is removed:

EURO-DAC '96 with EURO-VHDL '96
0-89791-848-7/96 $4.00

if *List* ∈ *LIST*
then if *List* ≠ {}
then *LIST := tail (List)*

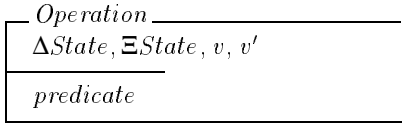
As in the case of *Z* (see below) it is assumed that the standard mathematical universes of booleans, integers, lists of any kind, etc. and all the standard operations on them are available. Further sophistication is introduced by *distributed EA-Machines*. They are given by a finite number of *modules* each of which is attached to a finite number of *agents*. Thus, a distributed EA-Machine can be seen as a set of concurrently running agents. Each agent is specified with the help of a finite set of transition rules. All these transition rules operate on a globally shared structure. In [BGM95] such distributed EA-Machines are used to define formally the semantics of VHDL93. As an example may serve the definition of the determination of the next point of time to be simulated:

if *AllProcessesSuspended*
thenif $T_n = T_c$
then *cycle := delta_cycle*
phase := update_driving_values
elsif *cycle = delta_cycle*
then cycle := postponed_cycle
phase := execute_postponed
else cycle := time_cycle
phase := update_driving_values
AdvanceTime
UpdateDrivers(T_n)
where T_n = *min*{*mindriver, mintimeout*}

This example shows how natural specifications in EA look like. It is described the following situation: If the expected next time T_n is equal to the current time T_c the kernel goes into cycle $delta\_cycle$. Otherwise the kernel goes either from $delta\_cycle$ to $postponed\_cycle$ or from $postponed\_cycle$ to $time\_cycle$. In the case of a $postponed\_cycle$ postponed processes are executed. In the case of a $time\_cycle$ the drivers are updated with respect to the new time T_n and T_c is advanced to T_n. When sufficiently refined, specifications in EA are executable. For this purpose runtime systems have been developed in Ann Arbor and Paderborn [DDG96].

1.2. The Specification Language Z

The non executable specification language *Z* [Sp94] is intended to define and reason about a wide range of systems. The *Z* notation consists of two parts: first of all the usual mathematical notation based on set theory and first order predicate logic is part of *Z*. Complete descriptions in *Z* are organized using schemata. Therefore a schema language for structural decomposition of descriptions constitute the second part of the language *Z*. In a certain sense *Z* follows an operational approach as there is the notion of *states* and *state transitions*. A state of a system is usually described as a number of variables and a number of associated invariant properties. An operation on the state that changes declared variables is defined by a schema like:



Change declared variables as defined in the pre and post-conditions (of the predicate)

By convention an operation schema that includes a state schema ΔState may change the values of the variables declared in *State*. Ξ*State* defines that the variables in *State* remain unchanged. An undecorated variable *v* appearing in an operation schema refers to the value of *v* before the operation occurs and *v'* refers to the value after the operation has been performed. Schemata may be composed using various *schema operators* like schema disjunction or schema conjunction. As most of today's engineers passed some education in set theory and mathematical notation only the relatively simple principles of the schema language have to be learned in addition to folk knowledge. This makes *Z* attractive as specification language. In fact, *Z* has been used as specification language in a couple of large projects in the hardware and software domain. The question is how to transform (non executable) specifications in *Z* to VHDL, say. The basic technique for this transformation is the refinement calculus (RC) as developed by Back [Ba88] and Morgan [Mo90]. This RC, initially defined for sequential specifications has been extended to support non-sequential embedded systems by Mahoney and Hayes [MH92]. Additional extensions cover real-time features and complex concurrent systems. In [Oc96] the system Z2VHDL is discussed. Z2VHDL takes a Z specification and refines it step by step into a version that automatically can be translated to executable VHDL. Obviously Z2VHDL is not a fully automatic procedure (in this case Z would have become an executable specification language) but a system that guides the user to apply refinment steps in such a way that a VHDL implementation is obtained. *Z* specifications on a sufficiently detailed level of abstraction, however, may be translated to VHDL automatically.

2. Visual Hardware Description Languages

2.1. Introduction

In a certain sense the trend in hardware specification and modelling during the past decade was a little bit paradox. As long as there was no good support for graphical computing available, graphical techniques (e.g. schematic entry) were widespread. When excellent support for graphics became available to affordable prices graphical techniques have been replaced by textual ones (VHDL). Obviously graphical techniques are not suited to replace all kinds of textual representations. There are good reasons that hieroglyphs finally have been replaced by letter based notations. On the other hand there are numerous situations where symbols and pictures allow a much more concise description, especially if pictures are put to motion. In our group we are experimenting with two powerful mechanisms which can easily bound to visual representations. The first one is given by high order Petri nets. For Petri nets there exists a very intuitive graphical representation. In fact most people have this representation in mind when they think in terms of Petri nets. We have enhanced the basic model in order to overcome most of the known deficiencies of Petri nets while preserving their benefits. The result has been called SEA (System Engineering and Animation). Parallel Logic Programming being a second paradigm for visual programming may be surprising. However this programming technique based on terms like agents, rules, messages can be represented in a very natural way in a visual form. An agent may have a graphical representations. It contains (also graphically) rules that define its behaviour. By an intuitive animation of message flow and pattern matching the behaviour of a parallel, logic programs becomes an easy to understand moving picture. In our case we follow the approach proposed by Kahn and Saraswat [KS90], called Pictorial Janus (PJ). In contrast to them we offer true online animation. Therefore our system is called Janus in Motion (JIM). Both of our approaches can be adopted to the specific pictorial world to be modelled.

2.2. Approaches Based on High Order Petri Nets

Since decades Petri nets play an important role in the area of system engineering. The original definition of these nets, however, turned out to be too elementary. To overcome these deficiencies various approaches of hierarchical nets and higher order Petri nets have been proposed. Here an extension of Predicate/Transition nets (Pr/T nets) will be discussed. Only a short informal introduction shall be given, where it is assumed that the reader is familiar with ordinary Petri nets. The first basic difference between these nets and Pr/T nets is that in Pr/T nets *tokens* are individuals while in ordinary nets only their cardinality on places is of interest. In contrary to ordinary Petri nets the *input arcs* of a transition are *labelled* with *typed variables*. A transition is *firable* only if there is a valid *interpretation* of the set of these typed variables using currently *instantiated tokens* in the respective places. Equally named variables attached to input arcs of one transition have to be substituted by the same values for an interpretation to be valid. Transitions have attached a *predicate* and a *token mapping*. A transition *fires* under a specific interpretation only if its predicate is true for this interpretation. So looking for a valid interpretation may be interpreted as looking for sufficient syntactically correct data while testing the predicate means testing whether semantical restrictions are met. If a transition fires it destroys the input tokens that constitute the interpretation it is reacting on, and calculates (i.e. instantiates) tokens on its output places. By labelling output arcs with typed variables values can be routed individually to token instantiations. Fig. 1 shows a Pr/T net transition before and after firing.

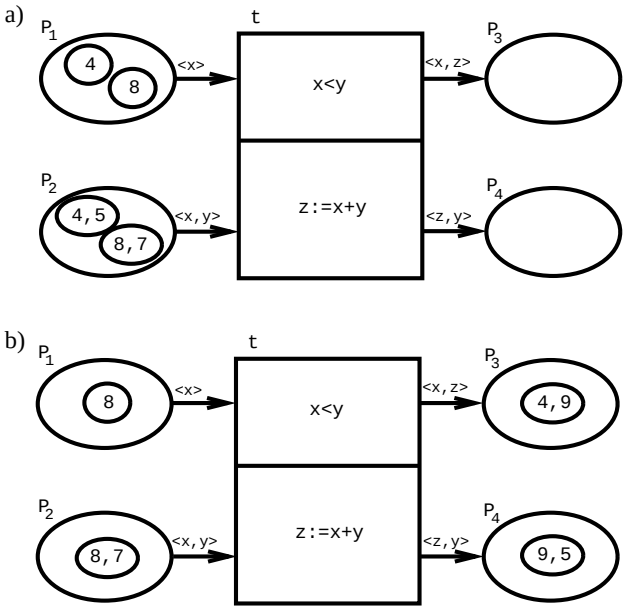


Figure 1. Pr/T net Transition Before and After Firing

In the case of our system SEA (**S**ystem **E**ngineering and **A**nimation) further annotations are introduced. As mentioned above, transitions in Pr/T nets are annotated with a condition and an action. This one may be seperated into preactions to be executed after the demarking of the input places and postactions to be executed right before the marking of the output places. SEA extends the basic concepts of Pr/T nets by including hierarchy and timing. Concerning hierarchy both, hierarchical (macro) places and hierarchical (macro) transitions are offered by SEA. Macro places have

a semantics inherited from StateCharts while the semantics of macro transition is inherited from Structured Petri nets [CK81]. Concerning timing, SEA offers *enabling delay* and *firing delay*. The first one defines the minimal time delay before an enabled transition becomes active while the firing delay defines how long it takes to fire a transition. In both cases *(min, typ, max)* specifications are possible. The actions attached to transitions may consist of computations of arbitrary kind and arbitrary complexity, formulated in any kind of language. By this also graphical modelling techniques can be incorporated easily by defining corresponding libraries. Pr/T nets have a standard graphical representation inherited from Petri nets. This representation, however, may not look natural to an engineer who is familiar with the graphical notations used in his or her domain. SEA therefore offers arbitrary graphical representations of *macro transitions* and *macro places*. By using the same interfaces as the corresponding standard representation these user defined pictures are bound to the semantics of the net in standard form in an unambiguous way. Of course for a dynamic modelling technique like Pr/T nets it is essential to animate also the dynamic behaviour of the non-standard representation. In SEA there is a rather simple, yet powerful mechanism to support animation. There may be an arbitray number of graphical objects associated to a macro-node. The macro-node has a vector of so-called *"visibility bits"*. Each bit corresponds to exactly one graphical object. Only such graphical objects are displayed that are enabled by the corresponding visibility bit. These bits may be manipulated by actions attached to transitions. The various graphical objects may differ in any aspect, even in shape and size. This allows a wide variety of visual animations. A small example, taken from [KKJ96] may serve to demonstrate these capabilities. Fig. 2 shows the general structure of a hierarchical description of an elevator system.

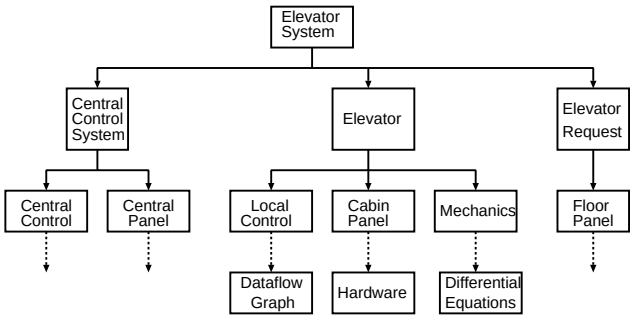


Figure 2. Elevator System Decomposition

It can be seen that various modelling techniques are intended to be used, ranging from data flow graphs to differential equations. Fig. 3 shows the top level graphics in SEA of this system for the case of four floors.

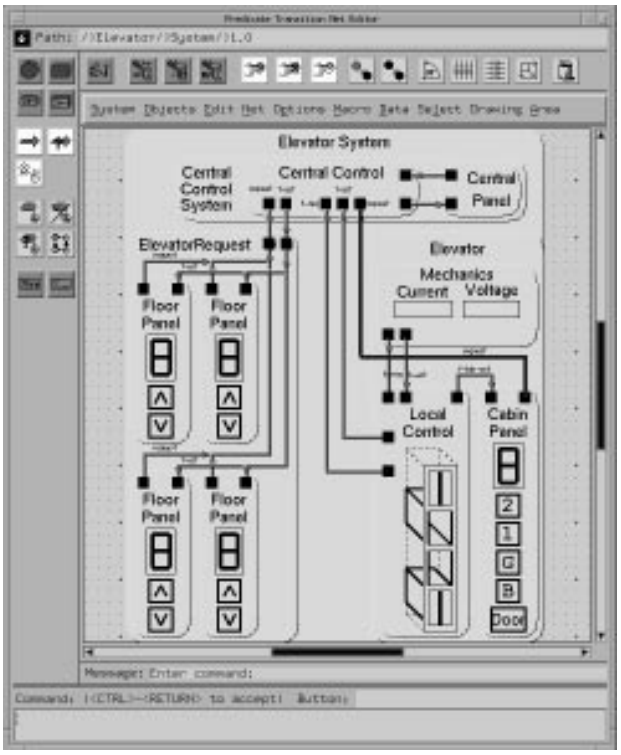


Figure 3. SEA Model of Elevator System

It should be mentioned that this is not just a picture but one possible graphical representation of an executable Pr/T net. As sound can be handled in the same way as graphics, the model used as an example here also produces all the sounds today's elevators tend to produce. The picture should be self explaning. Only the four graphical symbols within *Local Control* may need some explanation. They stand (top to bottom) for a closed floor door without cabin at this position, an open door with cabin, an open door without cabin and a closed door with cabin. These symbols are highlighted during simulation whenever the appropriate situation is true. If the model is correct, the third symbol is never highlighted, of course. This top level model has to be refined to make it operational and to hand over a more detailed model to implementation. So there are underlying Pr/T nets for all macro-nodes in the shown representation. This can be done also for subsystems to be modelled in a continuous way. This is needed to model the elevator mechanics in our example. Such parts usually are specified via a mathematical model using differential equations. In our example the elevator mechanics may be modelled as a mass system that is moved by an electric engine. This one is controlled by a controller that calculates the input voltage for the motor from the difference between the required elevation h_{reg} and the actual one h_{act}. Fig. 4 shows this

system.

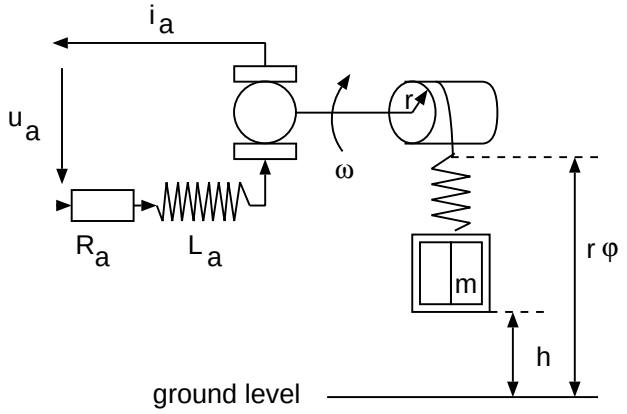


Figure 4. Elevator Mechanics

This situation can be modelled by differential equations that may be represented by the block diagram as shown in Fig. 5

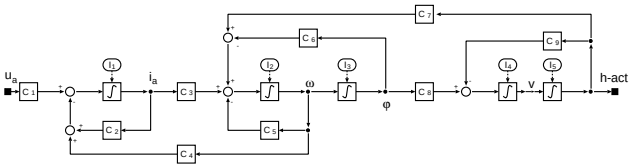


Figure 5. Block Diagram of Motor Driven Elevator

In [BR96] it is described how such block diagrams can be transformed into Pr/T nets using the Z-transformation. In SEA there is included a library for all elements used by mechanical engineers in block diagram representations of differential equations. Fig. 6 shows some examples from this library.

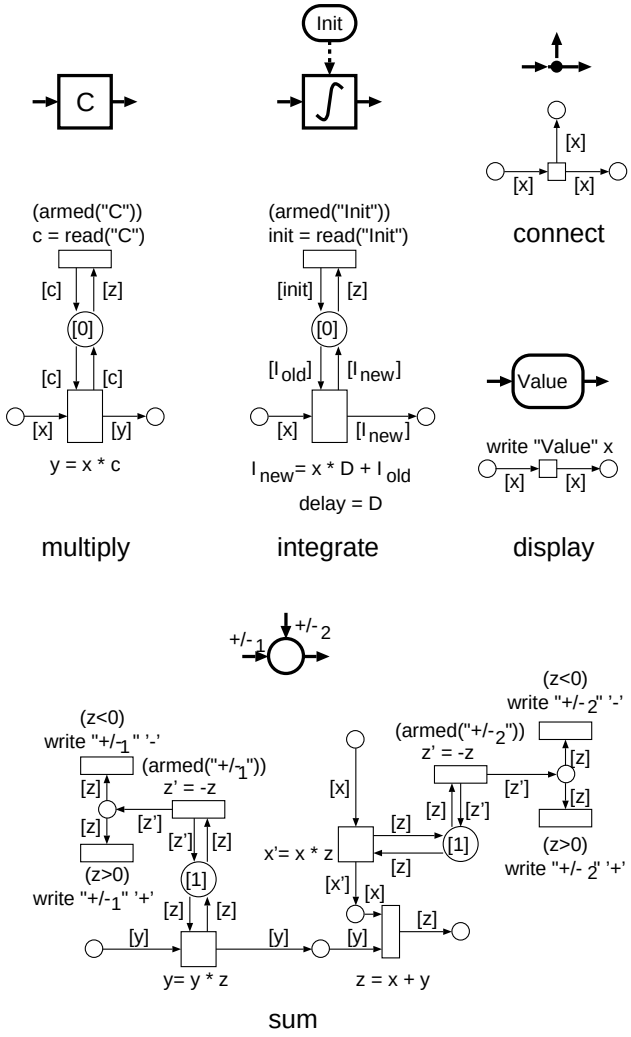


Figure 6. Library for differential Equations

SEA is a graphical specification and modelling system that covers the entire area of heterogeneous systems consisting of both, discrete parts and continuous ones. By its hierarchy concepts very complex situations can be handled. A user friendly interface allows various classes of users to work with SEA in the environment they are familiar with.

2.3. Approaches Based on Parallel Logic Programming

Concurrent logic programming has been investigated as system description language since a couple of years [WS87]. Here we shall concentrate on their pictorial version, more precisely on *Pictorial Janus (PJ)* and the online programming environment for PJ called *JIM (Janus in Motion)*. This has been developed in our group while PJ originates from Xerox PARC. Just to give a better understanding of the

principles of this pictorial language, which is based on concurrent logic programming, some basics of *Flat Concurrent Prolog (FCP)* as the most general representative of this class of languages shall be introduced here.

FCP is a general purpose logic programming language designed for concurrent programming and parallel execution. It has been developed at Weizmann Institute of Science in 1985 [MTSLS85]. The computational model of FCP is based on the *process interpretation* of logic programms in which active parts of a computation are conceived as concurrent processes. These concurrent processes communicate via *shared logical variables*. An individual process is represented by a *goal atom* of the form $p(A_1, A_2, ..., A_k)$. By this a process of *type p* with *arity k* and the *argument list* $A_1, ..., A_k$ is identified where the $A_1, ..., A_k$ are arbitrary *logical terms*. A process can perform one single operation, called *process reduction*. It is determined by the current values of the process' arguments, when and how the process reduction can be executed. Given a logic program, the behaviour of a process $p(A_1, ..., A_k)$ is defined by the finite subset of program clauses for predicate p and arity k. This subset is also called a *process procedure*. Each clause represents a rewrite rule for *goal atoms*. It has the structure of a *guarded Horn clause* of the form $A \leftarrow G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n$. Here A stands for the clause goal, G_i for *guard predicates* and B_j for *body predicates*. The guard predicates G_i states conditions referring to process argument values. In order to perform a reduction on a process A' using some program clause $A \leftarrow G_1, G_2, ..., G_m \mid B_1, B_2, ...B_n$, the goal atom A' has to *unify* with the clause's head atom A (as in ordinary PROLOG) and in addition all guard predicates $G_1, G_2, ..., G_m$ must be true under the selected unification. The body part $B_1, B_2, ..., B_n$ here is interpreted as a collection of atoms defining a multiset of concurrently active subprocesses. As result of a successful reduction step, these subprocesses spawn a local subnetwork that replaces the reduced process. The global network state is updated due to the unification carried out. A *process reduction* is possible if the related process procedure contains at least one enabled clause (unifyable and fulfillable guards). All clauses of such a procedure are tried in parallel (OR parallelism). In the case of multiple applicable clauses a selection is made indeterministically under control of the *commit operator* " | ". A clause is definitely selected for reduction after its commit operator has been processed. All other concurrently regarded clauses for this reduction are discarded at this time. The commit operator is irreversible, i.e. there is no backtracking.

Pictorial Janus (PJ), defined by Kahn and Saraswat [KS90] now is a complete visual programming language based on the parallel logic programming language *Janus*. Janus is a language which can be characterized as a simplification of FCP. These simplifications need not to be explained here. The sequel introduction to PJ and our system JIM is an excerpt from [DLMT96]. The basic elements of PJ programs are graphical primitives like closed contoures and connection arcs. As the meaning of such closed contoures is independent from its geometrical representation and graphical context, objects may be modelled in the most appropriate way concerning colour, shape, size, etc.. The basic primitives are combined to objects by the topological relationships attachment and inclusion. Concerning objects, PJ offers *agents*, *functions*, *relations* and *messages*. Messages may be *constants* or *list elements*. Each object may have ports to allow connections to other objects to be established. Agents contain one or more *rules* that define the agent's reaction to external messages. Fig. 7 lists the various PJ objects. In this figure ports are filled grey just to emphasize their contoures.

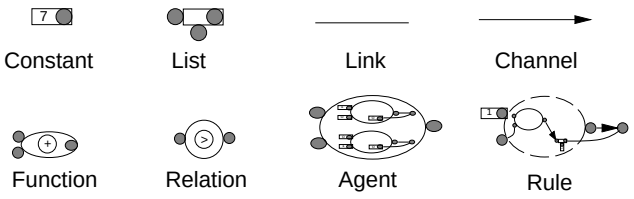


Figure 7. PJ Objects

Constants hold values while list elements establish more complex data structures. Different elements may be connected by *links* which are represented by unidirectional lines. They represent data dependencies. Functions and agents consume and produce *messages*. An agent (it corresponds to a procedure in FCP) is defined by a closed contour with a set of external argument ports. Its behaviour is defined by a set of rules which are located inside its contour. These rules correspond to clauses in FCP. A rule is basically a copy of the agent's interface. Each rule defines the behaviour of an agent with respect to different input patterns (i.e. guards). The guards are located outside the rule's contour wheras the behaviour *(subconfiguration)* is defined inside. A subconfiguration defines a set of linked objects, i.e. messages, functions and agents, being created when the rule is matched. Instead of directly specifying the behaviour of an agent, a *call arrow* may instantiate another agent or the agent itself recursively. A recursive call makes an agent *persistent* as it is replaced by itself on execution. *Channels* establish direct connections between two external ports. The meaning is to send messages to other agents. Fig. 8 shows a simple PJ program defining an AND-gate.

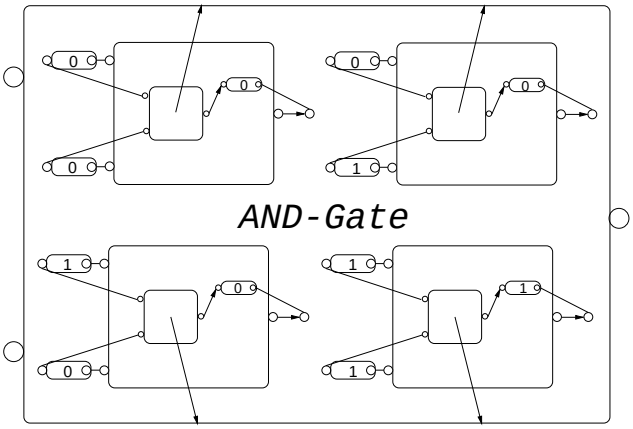


Figure 8. PJ Agent with 4 Rules

The exciting idea behind PJ is that it is not only a graphical representation of a logic programming language but that the visual animation of the program's behaviour is an integral part of PJ. So PJ completely follows the paradigm of *"draw a picture and look how it works"*. In order to understand this idea the execution model and its visual representation shall be discussed now. By a PJ program a net of communicating agents is given. These agents concurrently send and receive messages. In the case that an event has been detected at any port of an agent, i.e. a message has been arrived, the agent checks whether any of it's rules can be applied. This is done by simple *pattern matching*. The PJ execution model can be sketched by the following simplified steps. Assume that at least one new message has been arrived at one external port of agent *A*:

1. **Check rules:** *A* checks each of it's rules *r* for matching with the objects at the external ports. This phase is visualized later (see 2.).

2. **Select rule:** In the case that a rule *r* matches, *r* becomes a candidate for the further computation of the agent. One rule *r'* is nonteteministically selected from this set of candidates. Graphically now the objects within a subconfiguration of the selected rule are instantiated. Call arrows of agents are replaced by the corresponding behaviour. The selected rule continuously grows until its contour overlaps the agent's contour. The above mentioned pattern matching is visualized by morphing of the finally overlapping objects.

3. **Create subconfiguration:** The subconfiguration defined by *r'* is generated. This phase has already been visualized (see 2.).

4. **Link subconfiguration:** Objects of the generated subconfiguration are linked to the already existing configuration. As a result new messages are directed to agents which causes additional events to happen.

5. **Delete agent:** The agent *A* together with the matched input objects including their connections are deleted. Graphically the matched, as well as the matching objects smoothly disappear. The subconfiguration is resized until it fits to the space of the prevously deleted objects. The shrinking of links finally animates the motion of messages to their destination.

Fig. 9 gives a short animation sequence by six snapshots when matching a 0-0 combination at the input of the AND-agent of Fig. 8.

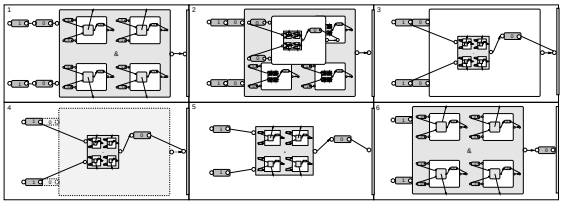


Figure 9. An Animation Example

In our research group we have implemented a programming environment for PJ with true online animation support, called *Janus In Motion (JIM)*. It consists of three main components: an *editor* serves as a graphical entry of the language. An *interpreter* executes the program on a logical level while an *animator* finally computes the visualization of the execution. The system can be installed on a distributed environment using PVM. Fig. 10 gives an impression of JIM's desktop.

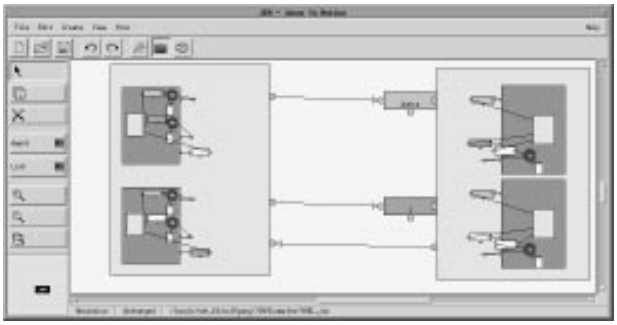


Figure 10. Screendump PJ Editor

2.4. Conclusion

VHDL will stay to be the dominating specification and modelling language in the area of hardware design for the

next years. In addition, however, various attempts are made to provide for more abstract, more formal (and by this formally verifyable) specifications. The two approaches presented in this paper are examples for this ongoing research. As both techniques are supported by very active scientific communities and as both have been applied to a variety of real application examples, these two approaches are certainly examples of specific relevance.

Certainly in the future visual languages will play a much more important role in any kinds of specification, modelling and programming. Again the two approaches presented in this contribution may serve as examples. Approaches based on Pr/T nets, e.g. SEA, have basic concepts in common with such ones based on parallel logic programming, e.g. JIM. There is a close similarity between the rules for activating a Pr/T net transition and activating a rule in a language like FCP or PJ. Both incorporate unification and guard-testing. For more details see [Ra95]. On the other hand hierarchy concepts from StateCharts and Structured Petri Nets have been included into SEA while at the same time SDL has been embedded into JIM [LMT95]. So a common understanding of pictorial modelling languages is evolving.

Combining these pictorial techniques with algebraic ones to a well understood and easy to handle multiparadigmatic approach to system specification and modelling may be the most promising solution.

2.5. Acknowledgements

This paper is an overview on ongoing work within the author's research group. Therefore original ideas and slightly modified excerpts from publications of the following persons have been used to compile this contribution: Egon Börger, Maria Brielmann, Giuseppe Del Castillo, Marita Dücker, Christian Geiger, Uwe Glässer, Bernd Kleinjohann, Lisa Kleinjohann, Georg Lehrenfeld, Wolfgang Müller, Christel Oczko, Jürgen Tacken, Christoph Tahedl.

2.6. References

[Ba88] R.J.R. Back: *A Calculus of Refinments for Program Derivations*.Acta Informatica,vol. 25,1988

[Bar95] D.L.Barton: *Overview of the SDDL Study Group*,June 1995

[BG94] E.Börger and U.Glässer: *A Formal Specification of the PVM Architecture*.In: B.Pehrson and I.Simon (eds.): Proc. of the IFIP World Congress'94, Volume I:Technology and Foundations,Elsevier Science Publishers B.V.,1994,pp.402-409

[BGM95] 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 (eds.):Formal Semantics for VHDL,Kluwer Academic Publishers,1995,pp.107-139

[BR96] M.Brielmann and F.J. Rammig: *Principles for the Development of Computer Based Systems*. In: Proc. of the IEEE Int.Symposium and Workshop on Engineering of Computer Based Systems,ECBS. March 1996,Friedrichshaven, Germany,pp.166-173

[CK81] L.A.Cherkasova,V.E.Kotov:*Structured Nets*,Lecture Notes of Computer Science 118,Springer,1981

[DDG96] G. Del Castillo, I.Durdanovic and U.Glässer: *An Evolving Algebra Abstract Machine*.In H.Kleine Büning(ed.)Proc. of Computer Science Logic -CSL'95,to appear in:LNCS,Springer-Verlag,1996

[DLMT96] M.Duecker,G.Lehrenfeld,W.Mueller and C.Tahedl: *A Distributed System for the Interactive Real-Time Animation of a Complete Visual Programming Language*.C-LAB Paderborn,Germany,C-LAB Report 05/95

[Gur93] Yuri Gurevich: *Evolving Algebras 1993: Lipari Guide*.In E. Börger (ed.):Specification and Validation Methods.Oxford University Press,1995

[KKJ96] B.Kleinjohann,E.Kleinjohann and J.Tacken: *The SEA Language for System Engineering and Animation.* Proc.ICAPN'96,1996

[KS90] K.M Kahn and V.Saraswat: *Complete Visualizations of Concurrent Programs and their Executions*.In Proc. of the IEEE Workshop on Visual Languages,IEEE,1990

[LMT95] G.Lehrenfeld,W.Müller and C.Tahedl: *Transforming SDL diagrams Into a Complete Visual Representation*.In: Proceedings of the IEEE Symposium on Visual Langauges'95, September 5-8,Darmstadt,Germany,1995

[MH92] B.Mahony and I.Hayes: *A case Study in Timed Refinement: A Central Heater*.In: Z Forum 1991,University of Queensland,January,1991

[Mo90] C.Morgan: *Programming from Specifications*, Prentice Hall International,1990

[MTSLS85] C.Mierowsky,S.Taylor,E.Shapiro,L.Levi and S.Safra: *The Design and Implementation of Flat Concurrent Prolog.* Technical Report CS 85-9,Dept. of CS, The Weizmann Institute of Science,Rehovot,Israel,1985

[Oc96] Chr.Oczko: *Embedding VHDL into a Formal Specification Environment - A Case Study.* In Proc. SIG-VHDL Spring'96 Working Conference,Dresden,Germany, Shaker Verlag,1996

[Ra95] F.J. Rammig: *Models and Tools for Integrated System Design*.In: Proc. of The 10th Congress of the Brazilian Microelectronics Society and 1st Ibero American Microelectronics Conference.pp.191-210,1995

[Sp94] J.M.Spivey: The Z Notation: *"A Reference Manual"*, Prentice Hall International,Series in Computer Science, 2nd ed.,1994

[WS87] D.Weinbaum and E.Shapiro: *Hardware Description and Simulation Using Concurrent Prolog.* Proc.IFIP CHDL87,North Holland,1987