# Visual Specification, Modeling, and Illustration of Complex Systems

Christian Geiger*, Georg Lehrenfeld**, Wolfgang Mueller*
*C-LAB, **Heinz Nixdorf Institut
Fuerstenallee 11, 33102 Paderborn, Germany

## Abstract

*This article introduces and discusses different innovative means for visual specification and animation of complex concurrent systems. We introduce the completely visual programming language Pictorial Janus (PJ) and its application in the customer-oriented design process. PJ implements a completely visual programming language with inherent animation facilities. We outline the transformation of purely visual PJ programs into textual imperative programming languages. The second part of this article investigates animated 3D-presentations and introduces our novel approach to an animated 3D programming language for interactive customer-oriented illustrations.*

## 1 Introduction

In recent past, several languages, methods, and tools are labeled to be *visual*. In the context of programming *"visual"* can denote very different properties. So, Visual Basic is mainly a language for the programming of graphical user interfaces; Visual C++ denotes a window-based C++ programming environment; VisualAge can be seen a combination of both. In all of these approaches the program is still captured and manipulated in a textual form. This is different in so-called visual programming languages. In visual programming languages, basic programming constructs for entering and manipulating the program are graphical objects (tokens). Visual representations can cover all sorts of 2D and 3D presentations including integrated textual parts.

Considering todays design of complex systems we can identify various domain- and application-specific notations. In object-oriented software engineering there are numerous graphical notations. The most popular ones come from Booch, Coad/Yourdon, Jacobson, Martin/Odell, Shlaer/Mellor, and Rumbaugh [1, 5, 9, 12, 18, 16]. Meanwhile, one of the three leading experts (Booch, Jacobson, and Rumbaugh) have decided to combine their approaches to the Unified Modeling Language (UML) which has been adapted as an OMG standard by the end of 1997. In hardware design mostly table- and diagram-oriented notations are applied. In this domain there are presently no major standardization efforts so that we can expect that the UML standardization will also strongly influence this domain.

Considering all the different currently applied visual means none of them are well-applicable for very early phases in systems design like functional specification and systems prototyping. Moreover, none of them really support collaborative end-user oriented design, i.e., providing means which can be easily understood by an inexperienced customer so that he/she can give immediate feedback on the functional correctness of the specified system. This feedback is necessary to check for design errors in early design phases. At present, functional specifications are mainly textual descriptions in English prose which typically gives multiple interpretations. Currently, cooperative design is primarily achieved through prototyping with graphical user interfaces for software, physical mockups and virtual prototyps for mechanics, and costly animated 2D and pseudo 3D models for general systems.

We are aiming at an early collaborative development of the system designer with the customer through programs with animated end-user-oriented presentations. By unifying the means of the specification and the animation language we avoid errors when transforming the program into a customer-specific representation. For this, we apply the completely visual programming language Pictorial Janus which was developed by Kahn and Saraswat at Xerox PARC [10] and investigate enhancements in several visual and functional directions.

The remainder of this article is structured as follows. The next section investigates visual languages

for systems design. Thereafter, we sketch the ideal case of the collaborative customer-centered design process as well as Pictorial Janus' role for visual specification and animation in the context of this process. Section 4 introduces a strategy to translate functional Pictorial Janus specifications into imperative textual programs. Though 3D animation can be implemented by an advanced 3D animation library the coupling with the program still takes considerable time for more complex systems. In order to reduce this effort we have developed the 3D programming language SAM which overcomes these limits. SAM is briefly introduced in Section 5 before the final section draws the conclusion.

# 2 Visual Languages in Systems Design

For a long time it was argued that "a picture says more than thousand words". Recent works are more realistic since these 1000 words are mostly not the same for each viewer of one picture [14]. Generally, we can say that textual languages are better for precise and concise specifications whereas visual languages are more adequate for systems structuring and the representation of complex interrelationships [13]. More precisely, visual languages are best for the design of complex parallel systems with concurrently communicating subsystems. This is because these systems are multi-dimensional in various directions (space, time, views, etc.) so that they are best captured by a multi-dimensional representation like static/animated 2D/3D.

In general systems design visual representations are applied for documentation, specification, programming, and analysis. During specification visual notations are mainly used for sketching the initial structure or rough I/O behavior of a system. However, in practice the first functional specification is presently often still in textual notation. Diagrams are mainly used for annotating the text therein and not vice versa. In later design phases graphical means are mainly applied for structuring hierarchical designs in form of structure- or block-diagrams. Only few other visual means support data- and behavioral specification.

Visual notations in systems design are often a combination of diagram-, icon-, and form-oriented representations where most of them have a large diagrammatic part [19]. Digrams are often given as a (un)directed graph. Nodes and edges can be annotated by text or program fragments. Advanced notations are based on *hypergraphs* or Harel's *Higraphs* (Hierarchical Graphs).[8][1] In Higraphs, nodes are combined as sets to super-nodes. Graphically, they are drawn to be contained in the super-node. An edge from/to the super-node stands for the set of edges from/to all of its contained nodes. This allows very structured representation and often most efficient comprehension of graphs as it is shown in Figure 1. That figure gives two different representations of a 3-clique. Note here that this figure gives a Higraph example and not a description in StateCharts which is an application of Higraphs. StateCharts are frequently used in parallel systems specification and modeling (e.g., StateMate).

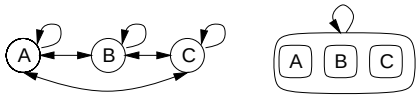


Figure 1: 2 Different Representations of a 3-Clique

The remainder of this section gives first an overview of diagrammatic means for behavioral specification. Beside tables, diagrams can be found as the most frequently applied basic visual notation in general systems design. In the next subsection we briefly discuss some UML (Unified Modeling Language) diagrams since UML gives a good unification of the different variations of most important diagrammatic means. Whereas UML presents the current state-of-the-art in visual systems design we give an outlook to the future by presenting the concepts of Pictorial Janus (PJ) thereafter. PJ as a completely visual language has been developed for handsketch drawing captures like pen-based devices. Moreover, once a pragram has been drawn its execution can be immediately inspected by animating the drawing through continuous motion of the individual objects.

## Unified Modeling Language

The Unified Modeling Language (UML) is the combination of different diagrammatic means with a methodology for object-oriented systems engineering [6]. UML defines development-, case- study-, package-, class-, activity-, sequence-, collaboration-, and state-diagrams. The following paragraph sketches the latter four which are for behavioral descriptions.

[1] Higraphs are inherited from Venn-Diagrams which are due to Euler's Euler-circles.

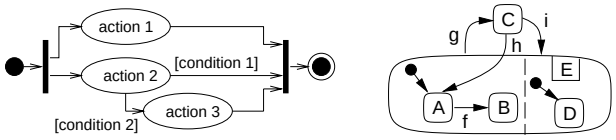


Figure 2: UML Activity and State Diagram

Activity-diagrams are a variant of classical control-flow diagrams. They give the imperative view of an algorithm. Nodes typically represent imperative statements. Parallel control-flows are supported by fork and join constructs. Figure 2 shows the example of a very simple activity diagram on the left. Black markers indicate the start and the end of the control-flow. The different conditions are given as labels at the different outgoing edges of a node.

Sequence-diagrams give the timeline-based interaction between parallel executing objects or processes. The line gives the execution order which starts at the top and advanced to the bottom of the diagram. Arrows between the time lines are labeled with methods which are called from the object or with messages which are sent. Figure 3 shows a very simple sequence-diagram in the middle with three communicating objects. Method invocations are given as solid arrows and messages are half arrows. The final termination of a process is given by an X at the end of the time line.

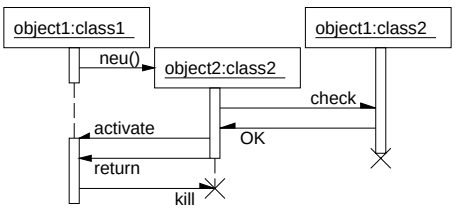


Figure 3: UML Sequence Diagram

UML state-diagrams are directed hierarchical graphs. Nodes define states or subgraphs. Edges define state transitions and are typically labeled by an expression of form $E(C)/A$. This defines that the transition is executed with action $A$ at event $E$ when additionally condition $C$ evaluates to true. This gives a so-called Mealy behavior. Assigning actions to states defines Moore behavior. Figure 2 gives a simple UML state diagram on the right. For the matter of simplicity we have labeled edges therein only by events. The figure shows the hierarchical state $E$ which covers 2 parallel state machines with states ($A$,$B$) and ($D$). The transition to their initial states is executed at event $i$. Event $g$ triggers the transition from all substates of $E$ to $C$.

We skip the introduction of the other diagrams and continue with PJ's concepts which give an outlook to possible directions in the future of visual languages. The reader who is interested in a more detailed UML introduction is referred to [6].

## Pictorial Janus

Pictorial Janus (PJ) is a complete visual programming language based on the parallel logical programming language Janus. PJ was introduced by Kahn and Saraswat at Xerox PARC [10].

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. Figure 4 gives the example of four different representation for one PJ message.

Figure 4: Different Representations

Basic PJ primitives are combined to objects by topological relationships (attachment and inclusion). For objects, PJ distinguishes *agents*, *functions*, *relations*, and *messages* (cf. Figure 5). Each object may have *ports* in order to establish a connection to other objects. Figure 5 gives a list of the various PJ objects. In that figure ports are filled grey in order to emphasize their contours.

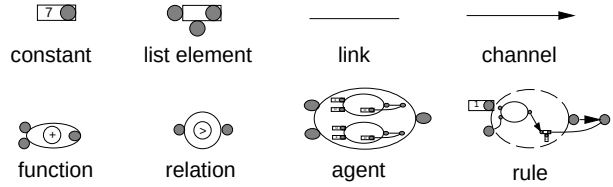


Figure 5: PJ Objects

*Constants* and *list elements* are denoted as messages. Constants hold values. List elements compose 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

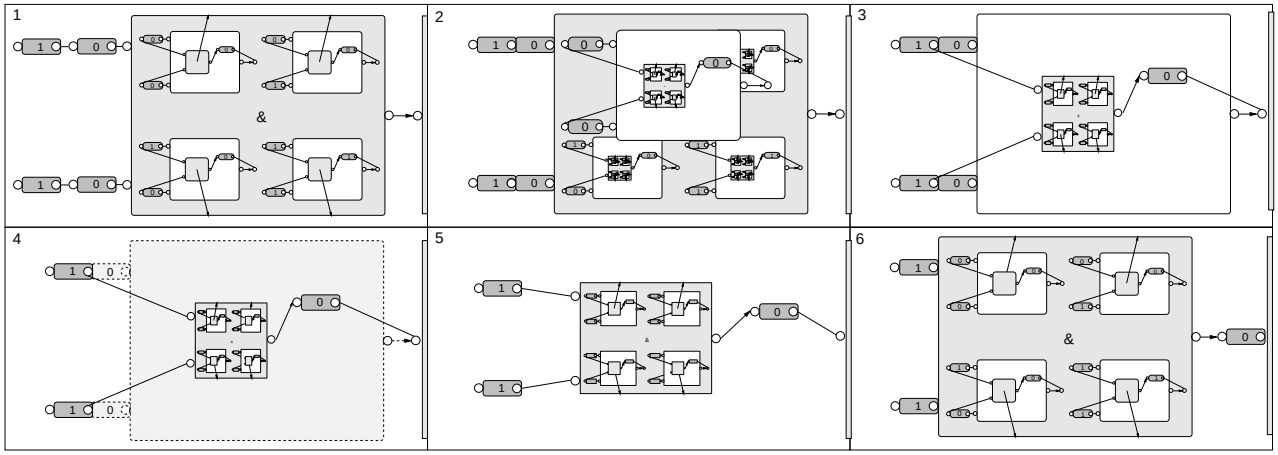

Figure 6: PJ Animation

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.

We briefly outline PJ by a simple example defining a logical AND in Figure 7. An AND can be defined as a function mapping input values to an output value. For a binary AND with 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 four cases, the rule creates the new configuration with an output value (message). The new configuration is specified as the recursive replacement of the agent.

Whereas the static PJ program is a drawing its execution is defined as a continuous animation. PJ is one of the very few visual programming languages which

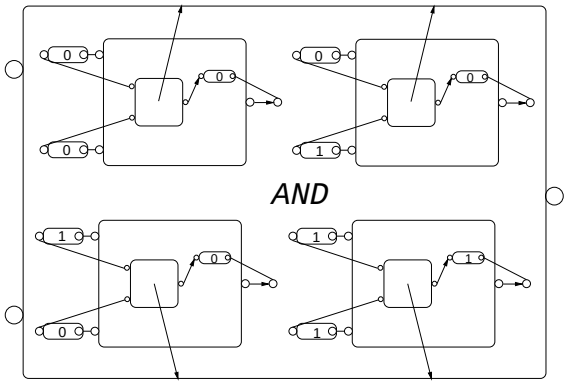


Figure 7: Logical AND

visualizes the detailed cause-effect relationship of each execution step through a continuous animation where each intermediate frame reflects a valid PJ program. The PJ execution model is based on the principles of pattern matching. In more details, the animation can be summarized by the following four steps.

1. Conditions of rules are checked against the messages at the input of the agent. One rule which evaluated to true is being selected.
2. The selected rule including its condition and body objects is enlarged to the size of the agent until its condition overlaps the messages at the agent's input.
3. Rule including condition and matched messages disappear in thin air. The objects in the rule's body are kept.

4. The body objects are enlarged to their default sizes.

As an example, Figure 6 gives the animation of the program in Figure 7.

# 3 The Collaborative Systems Development Process

Each individual phase of the systems design process can be roughly divided in specification, implementation, and analysis. Due to the properties of the system under design we can identify other subphases like partitioning (e.g., into hard- and software). Requirement analysis and specification take place in the very early phase of systems design. This phase requires a most frequent communication and information exchange of the designer with the customer in order to agree to a first contract. With this contract the designer tries to implement the required system. Practical experience shows that is makes sense to contact the customer from time to time in order to minimize the deviation from his/her requirements or wishes. Deviations can be due to different interpretations, conflicting specifications, or not implementable specifications. To avoid this, Repenning proposes a model for a collaborative development in [15] which integrates the customer's feedback in the very early phases of the process. Figure 8 shows the modified form of this model.

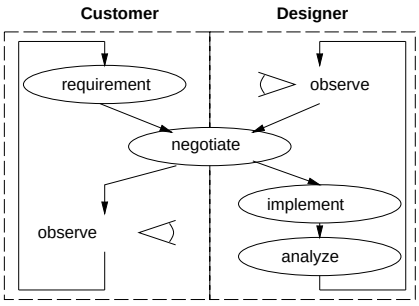


Figure 8: Collaborative Systems Design

The model is based on the principle of bidirectional observation. First, the designer tries to understand the customer's requirements. The following negotiation reduces these requirements to the technical realizable subset. This can be seen as the first contract between designer and customer. In the ideal case the customer observes the designer during implementation and analysis and collaboratively tries to refine the requirements. In practice, the biggest problem is that it is mostly not possible for a customer to understand the details of the implemented functions. The designer often faces the problem to communicate the details of a function to a customer. This is mostly done by the means of graphical user interfaces which partly emulate the required function as it is understood by the designer. When this is not sufficient often more advanced techniques from algorithm animation are applied [2]. For algorithm animation the program code is annotated by additional commands for visualizing specific, so-called interesting events. This can be done, for instance, by the insertion of print statements and linking the output to the triggers of a graphical animation library. However, this technique is very time consuming and often requires skills in the application of a separate programming language or animation library. Therefore, the collaboration with the customer is often reduced to the minimum of the absolutely necessary interactions.

Our approach is based on the application of a visual programming language to support collaborative design in early design phases. We take PJ in the presented approach since PJ combines functional means for system specification with means for defining its animation in customer-oriented illustration. So, the designer is not enforced to make costly and error-prone translations between different languages or representations. The next chapter outlines how to integrate PJ into the first steps of the todays mostly still text-oriented design process.

# 4 Visual-Textual Design

Structured top-down design is based on the principle of stepwise refinement. In the ideal case the functional specification defines what to design whereas the implementation finally describes how it is designed. In the ideal case, the first executable specification is written by declarative means focusing on functional I/O behavior at the interface. The final implementation is typically written in a imperative programming language like C, C++, and Java.

We investigate a design process which incorporates visual 2D and 3D means for customer-oriented animated illustrations. Figure 9 gives an overview of the individual steps. The process starts from the purely visual means of Pictorial Janus where the drawings are stepwisely transformed into a textual imperative programming language. For advanced illustrations we integrate animated 3D after the initial visual specification.

In the remainder of this article we outline our con-

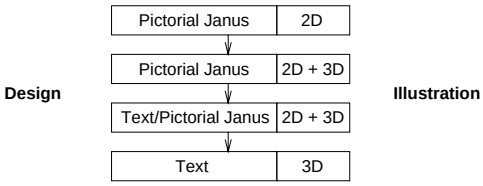


Figure 9: From Visual Specification to Animated Textual Implementation

cepts by a running example from automotive industry. A frequently investigated problem in the manufacturing production line is the dimensioning of buffers for scheduling internal objects. In our example we consider the color sorting assembly buffer for the enameling line of cars as sketched in Figure 10.

In order to avoid frequent and thus costly color changes in the enameling unit the raw car bodies are presorted and buffered by their requested color in a so-called color sorting assembly buffer. The bodies are sorted to blocks of the requested colors. There are typically less lines in the buffer than there are colors. The basic goal is to find a optimal strategy for minimizing throughput, color changes, etc. Since the problem is NP-complete more or less effective heuristics are applied in practice. Most of them make use of priority rules [17].

In our investigation we start the design process with an initial (executable) PJ specification. For most of the already specified structure in Figure 10 this means a 1:1 mapping to the network view of communicating PJ agents. Thereafter, the somewhere specified I/O relationships have to be transformed into PJ rules of corresponding agents. Figure 10 gives a first simple functional PJ program with two feeder lines at the lower part. The reader can easily see the correspondence to the structure given in the upper part of Figure 10. This first PJ program does not specify the scheduling strategy. It just gives simple I/O relations for building blocks with two colors. From left to right the program sorts red requests to the top and green ones to the bottom. The next agent collects 3 red requests and 2 green requests to one block. The rightmost agent models the enameling unit with labeled grey bodies at its input and colored bodies at its output.

The next step is to give the various agents and messages 2D representations which are more intuitive for the customer. So, messages can be given the form of a car as indicated in Figure 11. This is the moment where the first validation with the customer can be undertaken. With these more intuitive representations the designer can demonstrate the basic functional behavior of the system. The customers can observe the flows of the bodies through the different lines and give a first feedback. In some cases it even makes sense to integrate a 3D animation. The 3D model is then to be triggered by key events from the program. Figure 11 shows the integrated 3D animation for our example. In our system each message created at the output of an agent generates an event which triggers the motion of the cars in the 3D model. This gives a semi-automatic integration with 3D model as it is described in [3].

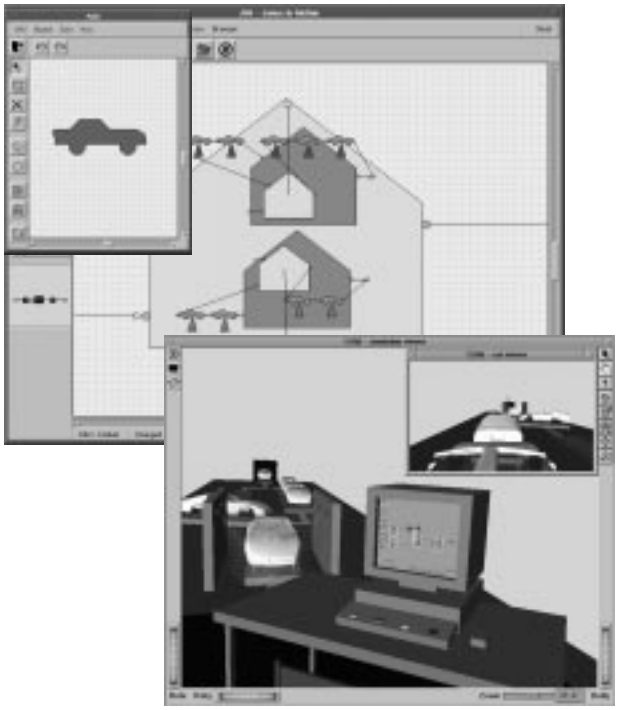

Figure 11: Different Customer-Oriented Representations

Thereafter, the previous functional PJ program is stepwisely transformed into an imperative algorithm. First, we have exactly to identify the interface of each agent. This is identifying the signature for each agent since untyped PJ agents have to be translated into typed functions. Second, the different rules of each agent are transformed into templates following the structure of embedded case statements. The individual templates hence have to be refined when replacing the abstract I/O function by algorithmic statements. The example in Figure 12 shows the combination of a visual with a textual agent. The left agent in this figure defines a much more simpler scheduling strategy than the right one does for the dislocation. Textual

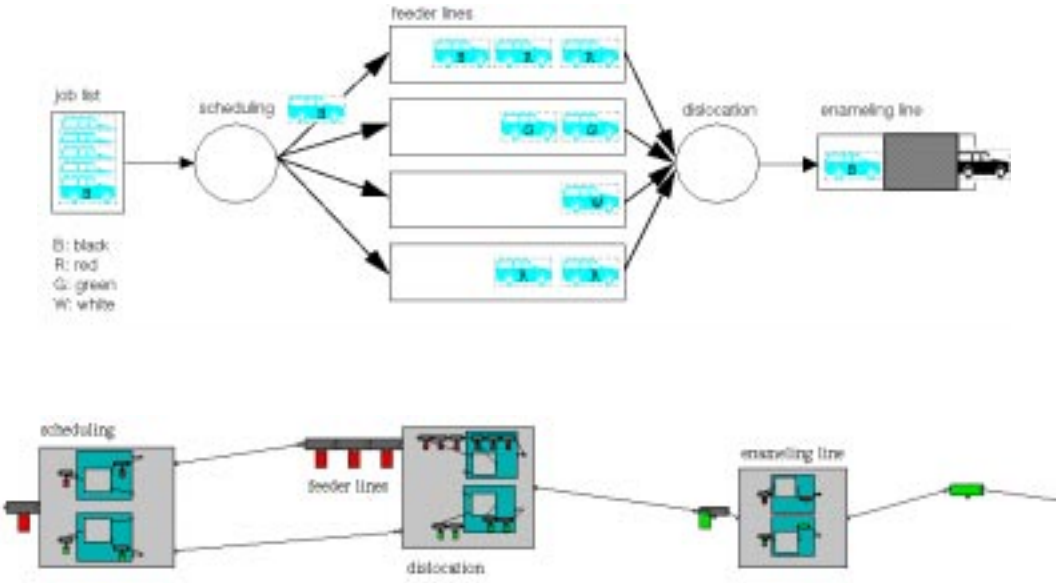


Figure 10: Color Sorting Assembly Buffer - Schematic Representation with PJ Programm below

descriptions can often define more complex behavior since state-oriented specifications, specifications with a large number of rules, or embedded rules easily leads to an overloaded visual representation which is hard to capture by inexperienced readers.

In a final step, the template of each rule including the precondition has to be completed by a textual (yet mostly imperative) program. Given the C or C++ programming language, for instance, the agent is roughly translated to a function with a SWITCH-statement where each rule translates to a CASE-statement in that statement. In next steps it is necessary to refine the expressions of the individual CASE-statements in order to implement the scheduling strategy.

The previously described transformation from graphics to text basically keeps the structure of the program. It is thus possible to keep the 3D animation as it was previously defined and to trigger it from the text without any significant changes in the program. This gives the customer a more homogeneous view on the general design without rapidly changing the model at the visual display. It easily allows the customer to observe the changes in the program's function through animation without getting lost in detailed programs.

It is important to mention here that the previous transformation basically includes manual transformations. Nevertheless, these transformations are mappings between well-defined structures and thus give the possibility to get managed by a semi-automatic process.

## Implementation

Our investigations have shown that the animation of single execution steps only makes sense in a highly interactive animation environment. For this we have developed the interactive PJ specification and animation environment JIM (Janus In Motion) [11]. Investigations have shown that animation can only be usefully applied when fast rewind, record, slow motion in combination with classical debugging functionalities like breakpoint, single step, etc. is supported. Additionally the environment has to support the direct manipulation of the objects in the animated representation so that single objects can be easily created, deleted, and modified at any time.

Since JIM was limited by its 2D representation in a few specific applications we have extended JIM by integrating 3D animations. For this we have used our OpenInventor-based 3D animation library [7]. Nevertheless, with this integration the designer again has to manage additional different means for 3D modeling and programming of application-specific animations. This typically results in extra effort in addition to the implementation of the program. This effort is reduced by the animated 3D programming language in the next chapter.

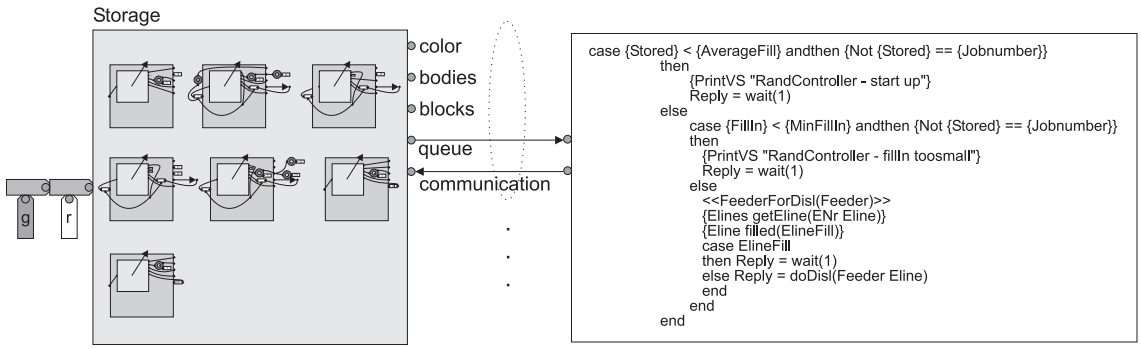


Figure 12: Complex Strategy in Combined Visual and Textual Form

# 5 An Animated 3D Visual Programming Language

Our recent results show that the means for implementing the system and the means for defining the 3D animation can be combined to one 3D programming language so that the designer does not need to learn two languages, one for the program and one for the customer-oriented 3D animation. The result is our new approach to the first animated 3D programming language SAM [4]. In contrast to other 3D programming languages which support the generation of 3D animations in our visual 3D programming language the main tokens are given as 3D objects.

## 5.1 Concepts

SAM (Solid Agents in Motion) is a visual 3D programming language based on the improved and enhanced execution model and concepts of Pictorial Janus. The execution model reduces several drawbacks of Pictorial Janus. A SAM program is a set of parallel state-oriented agents communicating by exchanging messages. The behavior of an agent is defined by a set of production rules which execute actions. An action can modify the agent's local attributes (e.g., state, size, color, location), execute local operations (e.g., computation of simple arithmetic expressions, rotation, play MPEG), send messages, or kill and spawn agents. An agent communicates via its input and output ports. SAM supports groupcast and broadcast for communication. A SAM production rule has a condition and a sequence of actions. Conditions include checks for current state as well as types (integer, string, coordinate) and values of messages at input ports. If the conditions of a production evaluate to true the agent executes the list of actions in sequential order. SAM is a synchronous parallel programming language. That means, that actions immediately produce messages at the output of an agent but these messages are kept at the output until all agents have finished the execution of their actions.

## 5.2 Visual Representation

In its abstract representation a SAM agent is given as a transparent sphere. Its color visualizes the state of the agent. Ports are little cones attached to its outside. The different colors of the ports allows to compare them with their formal representative in the rules. Production rules are located inside the sphere. A rule is basically a copy of the agent inside which the sequence of actions is given as a tube with the individual actions as its slices. A larger flat slice, the scanner, indicates the first action in the list. Different types of actions are given in different colors. An individual action is specified in a textual form. The textual part of an action appears when moving the mouse pointer over visual representation of that action. Messages are similarly defined. Figure 13 and 14 gives the different abstract representations of SAM objects.

The user can assign concrete non-transparent representations to agents and messages by simply providing an external OpenInventor file with the identifier of the agent as its name. Concrete and abstract representation can be exchanged at any time during execution.

## 5.3 Visual Execution

The visual execution of the program is divided into three phases. (i) rule checking, (ii) action execution, and (iii) message exchange. Like in PJ, after matching the rule, the selected rule is enlarged to the size of

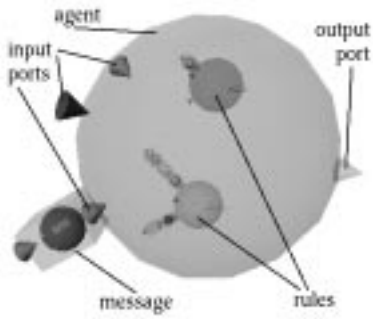


Figure 13: SAM Agent with One Scheduled Message

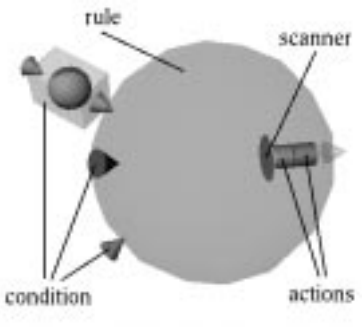


Figure 14: Enlarged Rule of an Agent

the agent. Unlike PJ, the scanner then moves over the actions indicating the currently executing action. In the case of a send-message action the message appears at one of the agent's output ports. For groupcast and broadcast, there is one message generated for each receiver. After the execution of the last action the rule is reduced to its former size. After all agent's have executed their last action messages move to their destination object.

Our SAM editor and our SAM animation environment is implemented on SGI as described above.

### 5.4 Example

We finally sketch a simple SAM program. Consider the Mars mission scenario with communication between path finder, sojourner, and earth. Earth sends a request to sojourner via path finder (the space probe) for taking a photo. In SAM, sojourner changes to status *active* for taking a photo and performs five actions: moving to the specified object (translation), taking a photo (sending a message), returning (rotation, translation, rotation). Pathfinder simply forwards the photo in form of the message to the earth. Figure 15 gives the simple SAM program with the abstract representation and the concrete representation below.

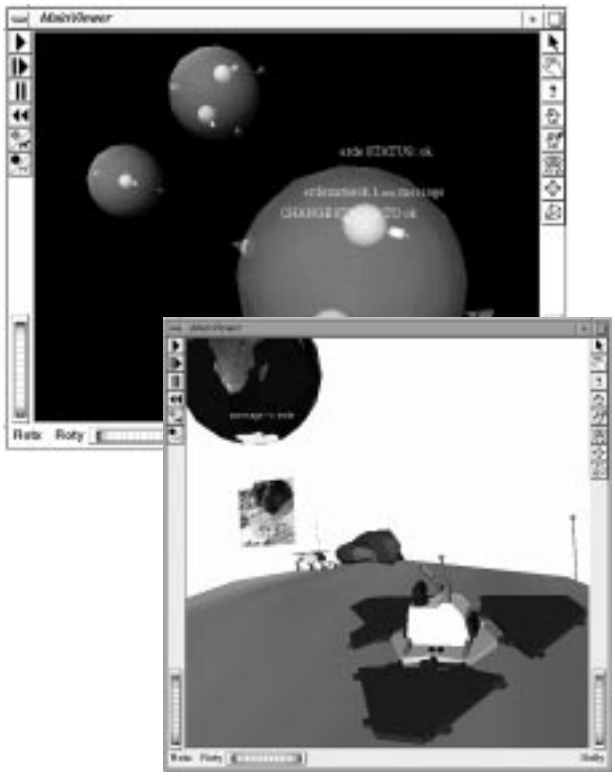

Figure 15: Abstract and Concrete Representation of a SAM Program

## 6 Conclusion

We have presented an approach for application of combined visual-textual means in the collaborative customer-oriented design process. For initial functional specification we use the complete visual programming language Pictorial Janus. Pictorial Janus supports customer-oriented static representations and animations for capturing the system's function. Pictorial Janus helps to avoid the usage of different means for defining the program and for defining the animation. We have sketched the transformation of a declarative Pictorial Janus program to an imperative textual program. Finally, we introduced the new concepts of the 3D programming language SAM. Like PJ, SAM combines the means of a programming language with those of an animation language. SAM reduces some of PJ's main drawbacks like temporal objects, lack of sequential operators, complete visualization of com-

munication, and no groupcast/broadcast.

So far we have performed no detailed user studies applying the presented approach. However, first experiments with our environment have shown that it is by far easier to explain programms when they come with a user-oriented animation. It greatly helps for outlining the function when the animation is triggered from the program so that the correspondence between program and animation can be easily seen.

Despite all of PJ's great advantages we are aware that our concepts are based on a pretty academic approach. This is mainly since PJ belongs to the class of parallel logic programming languages which are not very popular in todays systems design. Present industrial implementations are mainly in Fortran, ADA, C, C++; Java is emerging. Nevertheless, we have shown with SAM that with some small but significant modifications it is easy to achieve a synchronous programming language with the same flexibility in visual representation and animation. Synchronous languages, e.g., Esterel, Signal, Lustre, StateCharts, are frequently used for todays reactive systems design. Embedding those languages in a visual specification and animation framework based on the concepts of Pictorial Janus will surely give new directions in general systems design. We hope that this article is a first step in this direction.

## Acknowledgements

The work presented in this article is a combination of several subprojects. We would like to thank Marita Duecker, Ralf Hunstock, Waldemar Rosenbach, and Christoph Tahedl for their contribution in the implementation of some parts of the herein presented systems.

# References

[1] G. Booch. *Object oriented analysis and design with applications.* Benjamin/Cummings Publ. Co., 1994.

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

[3] Ch. Geiger and R. Hunstock and G. Lehrenfeld and W. Mueller and J. Quintanilla and A. Weber. Visual Modeling and 3D-Representation with a Complete Visual Programming Language- A Case Study in Manufacturing. In *IEEE Symposium on Visual Languages*, Boulder, Colorado, USA, October 1996.

[4] Ch. Geiger and W. Mueller and W. Rosenbach. SAM - An Animated 3D Prohgramming Language. In *IEEE Symposium on Visual Languages*, Halifax, Canada, September 1998.

[5] P. Coad and E. Yourdon. *OOD: Object-Oriented Design.* Prentice Hall, 1994.

[6] M. Fowler. *UML DISTILLED.* Addison-Wesley, 1997.

[7] C. Geiger. Prototyping interactive animated objects. In B. Preim P. Lorenz, editor, *Simulation & Visualisierung*, Magdeburg, March 1998. SCS.

[8] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[9] I. Jacobson. *Object oriented software engineering.* Addison-Wesley, 1995.

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

[11] M. Duecker and G. Lehrenfeld and W. Mueller and C. Tahedl. Specification and Analysis of Concurrent Systems in a Complete Visual Environment. In *European Simulation Multiconference (ESM96)*, Budapest, Ungarn, 1996.

[12] J. Martin and J.J. Odell. *Object oriented methods : a foundation.* Prentice Hall, 1995.

[13] W. Mueller. *Executable Graphics for VHDL-based Systems Design.* PhD thesis, Universitaet-GH Paderborn, Nov 1996.

[14] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 37(10), 1995.

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

[16] J. Rumbaugh. *OMT insights.* SIGS Books, 1996.

[17] S. Voss, S. Spieckermann. Simulation von farbsortierspeichern in der automobilindustrie. In *Simulation und Animation fuer Planung, Bildung und Praesentation*, Magdeburg, 1996.

[18] S. Shlaer and S.J. Mellor. *Object oriented systems analysis : modeling the world in data.* Yourdon Press, 1988.

[19] N. C. Shu. *Visual Programming.* Van Nostrand Reinhold, New York, 1988.