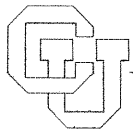
Creating User Interfaces with Agentsheets *

Alex Repenning

CU-CS-517-91



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was funded by the United States Bureau of Reclamation through the Advanced Decision Support program.

Creating User Interfaces with Agentsheets[1]

Alex Repenning

CU-CS-517-91 February 1991

Department of Computer Science
Campus Box 430
University of Colorado at Boulder
Boulder, Colorado 80309-430
Telephone: 303-492-1592
Fax: 303-492-2844

This research was funded by the United States Bureau of Reclamation through the Advanced Decision Support program.

---

[1]This report is a preprint of a paper to appear in the ACM/IEEE proceedings of SAC'91, Kansas City, MO

Creating User Interfaces with Agentsheets

Alex Repenning

Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder CO 80309
303 492-1218, ralex@boulder.colorado.edu

Abstract

Building user interfaces with tool boxes has limitations. Most of them allow users to build basic interfaces efficiently on a high level of abstraction. However, when users have needs for new building blocks, which cannot be composed of existing ones, users have to fall back on the low level of abstraction provided by conventional programming languages. Agentsheets address this problem by introducing an intermediate level of abstraction between high-level building-blocks and the level of conventional programming languages. They provide means for the incremental definition of the behavior as well as of the "look" of artifacts. This paper gives a short introduction to Agentsheets and then elaborates the concepts in form of a case study describing the process of building a new front-end for a commercial expert system with Agentsheets.

1. Introduction

Agentsheets, similar to spreadsheets, are based on a grid structure. The elements of the grid are called *agents*. Every agent represents its state in form of a graphical *depiction*. The agglomeration of all agent depictions results in the graphical representation of an artificial world. Figure 1 delineates a simple simulation of a Turing machine consisting of agents modelling pieces of tape, the head moving on the tape, and a gas container.

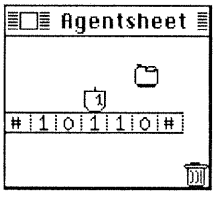


Figure 1: "Turing Machine" Agentsheet

Users interact with the Agentsheet by operating on agents with the mouse (e.g., clicking, dragging). For example the Turing machine gets started by dragging the gas container onto the Turing machine head. The Turing machine gets "programmed" by dragging new tape and head agents into a second Agentsheet representing valid transition.

The intended application domain of Agentsheets is *visual programming* [13]. The definition of semantics in the Agentsheet paradigm is based on *spatial reasoning* primitives like *neighborhood* and *distance*, e.g., the Turing machine head perceives the tape by checking the depiction of the tape agent immediately below it. Spatial reasoning can be used to express higher-level concepts like data-flow [6, 18] flowcharts [4], Nassi-Shneiderman structure diagram (NSD) [4], and augmented transition network (ATN) [4] to name just a few of them. We do not consider Agentsheets to be a visual programming system by itself. Instead, we think of Agentsheets as a tool that helps one to build visual programming systems.

Visual programing cannot completely replace conventional programming languages. Many abstract conceptions can be pleasingly represented by arranging icons [1], plotting data-flow, etc. Nevertheless, it is our strong belief that there will always be situations in which a textual representation is more concise and more desirable than a graphical one. This is especially the case if a visual programing system is only a syntactic variant of a conventional programming language.

In situations in which a graphical representation is inadequate, either because the pure-graphics approach would be very long-winded or the set of *building-blocks* provided is too restricted, a user will be forced to resort to programming on a much lower level of abstraction. The step between a building-block level and the level of a conventional programming language used to implement the building-blocks is what we call the "Representation Cliff." These users not only have to understand the underlying programming language, they also have to know about the possibly very complex transformation between the language constructs (e.g., a library consisting of a large set of functions) and the graphical representation.

Agentsheets provide a higher level of fall-back than just-graphical-building-block based approaches by introducing a

spatial reasoning layer between the building-block level, representing graphical primitives, and the conventional programming level (Figure 2). This does not completely eliminate the need to access the conventional programming level by the user, but it does allow users to resort to a higher level of abstraction in most cases.

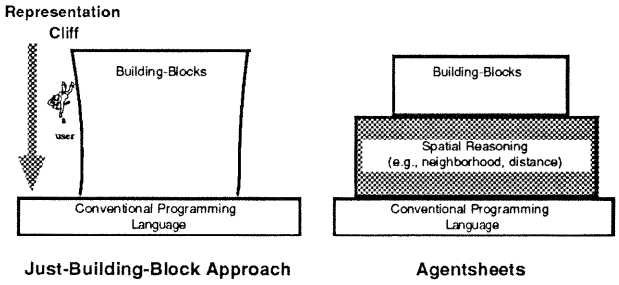


Figure 2: Levels of Abstraction

2. The Agentsheet Paradigm

Agentsheets are based on principles of object-oriented programming [5, 11] as well as on graphical extensions of spreadsheet programming [7, 10, 16]. Details regarding the use of Agentsheets are deferred to the case study section.

2.1. Agents

An agent [3] is a thing (or person) empowered to act for a client. The client, in turn, can be another agent or the user of the Agentsheet. Every agent consists of:

- *Sensors.* Sensors are *methods* of the agent that are either actively triggered by the user (e.g., clicking at an agent), or that are used to poll other agent's state. The built-in classes of agents provide a default *behavior* defining reactions to all sensors. In order to refine this behavior, sensor methods can be shadowed or extended making use of the object-oriented paradigm.
- *Effectors.* A mechanism to communicate with other agents by sending messages to agents using relative or absolute grid coordinates. The messages, in turn, activate sensors of the agents to be effected. Additionally, effectors also provide means to modify the agent's depiction or to play sounds.
- *State.* Describes the condition the agent is in.
- *Depiction.* The graphical representation of the state, i.e., the *look* of the agent.
- *Instance-of.* Link to the class of the agent.

2.2. Galleries

The incremental construction approach of Agentsheets is not limited to the behavior of agents. A tool called *gallery* allows the incremental composition of depictions as well. The gallery serves the following functions:

- *Clone depictions.* A new depiction in the gallery is created by *cloning* an existing one. In the simplest case, cloning involves only copying. However, cloning might include an additional transformation called the *cloning operation.* The set of cloning operations currently contains: unary operations (copy, rotate multiples of 90 degrees, flip horizontally or vertically, inverse) and n-ary operations (and, or, x-or). The gallery not only shows the depictions, it also makes the relationships among the depictions explicit; what is a clone of what (for an example see Figure 5).
- *Re-clone depictions.* Modification of a depiction can be propagated to the dependent depictions by re-cloning them.
- *Palette.* Instantiation of agents. The gallery acts as a palette from which depictions can be chosen and dragged into an Agentsheet.
- *Edit depictions.* A depiction consists of a bitmap and a name which can be edited with a *depiction editor.* The depiction editor is just another Agentsheet in which each agent represents a single pixel of the selected agent's bitmap. These agents make use of their mouse-click sensors in order to flip their depiction from black to white or vice versa.
- *Save and load depictions.* The gallery is a database containing depictions and relations. The set of depictions can be stored to files and retrieved from files.
- *Link depictions to classes.* Every depiction is associated with an agent class. This link is used when instantiating agents.

3. Configuration Charts: A Case Study

The Agentsheet is a general purpose paradigm which has been used for visual programming, user interface prototyping, and simulation. However, rather than giving a shallow description of several different toy utilizations we chose to elaborate on Agentsheets by describing a real-world application in more depth.

KEN [14, 15] is an expert system predominantly used to *configure* small to medium sized power plants. The configuration task includes the selection of components from a very large repertoire. A repertoire usually consists of 5 to 30 binders. The composition of components is dependent on physical (e.g., mechanical, electrical) and financial constraints.

There are basically three KEN user types:

- *End-User.* A sales person employing KEN side by side with a client to configure a complete system. The

configuration is a very interactive process in which the sales person and the client explore many "what if..." scenarios to optimize the functionality/price ratio of a plant.

- *Knowledge Engineer.* Mediating between expert and expert system. The knowledge engineer has a very good knowledge of the expert system and a basic understanding of the problem domain.
- *Expert.* Providing the problem domain knowledge. Most experts are not programmers, i.e., they either do not have time to acquire programming experience or they are simply not interested in doing any programming.

Once a knowledge base for a certain type of plant has been created, configuring a new plant can be achieved by relatively inexperienced end-users. The *knowledge acquisition*, however, is less than satisfactory. The *situation model* [2], the conceptual model of the problem domain, employed by experts is very different from the underlying *system model* of KEN (see Figure 3). In order to bridge this gap a knowledge engineer is employed to transform the expert's conceptions into the system model (frames and demon functions in this case).

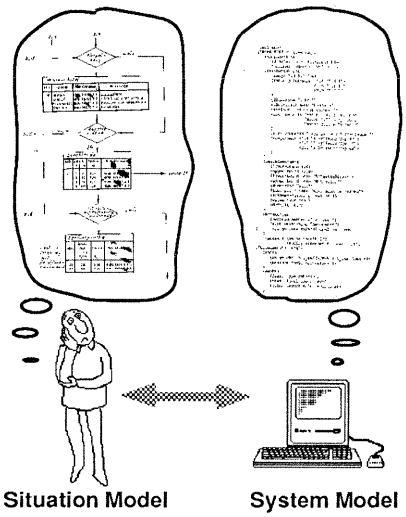


Figure 3: Situation Model vs. System Model

Although KEN has a powerful frame-based [9] knowledge representation, it does not provide a higher level of abstraction relating to the configuration domain. That is, the knowledge engineers have to "re-code" similar chunks of expert knowledge over and over again in the same way. Furthermore, it is almost impossible for an expert to *maintain* a given knowledge base because (a) the transformations made by the knowledge engineers are "compiled" into the knowledge representation, and (b) the text-based knowledge representation has a very strong lisp flavor and is, therefore, not very appealing to the experts.

3.1. Situation Model: "Natural" Knowledge Representation

Independent of any expert system based approaches, most experts in the field of configuration have developed their own paper-and-pencil knowledge representation schemes. Many of these experts express *procedural knowledge* (e.g., the sequence in which certain components can be constructed) employing a flow-chart like graphical representation. In the following, we will call this representation *configuration chart* (or CC for short). Figure 4 shows an actual CC drawn by an expert. Typical applications involve between ten and a few hundred of these sheets.

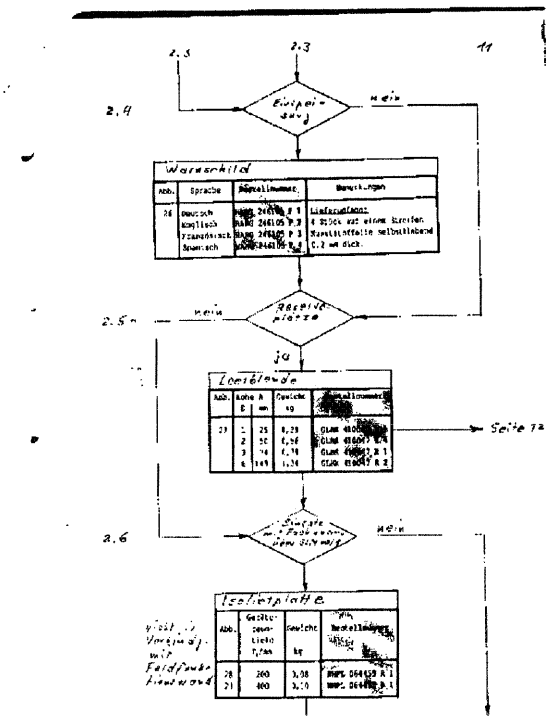


Figure 4: Chart used by an expert

An analysis of charts from different experts revealed a common set of CC primitives:

- *If-then-else branching*: Apply a relation to component attributes (e.g., height > width).
- *Sequences.* Execute a list of chronologically ordered actions.
- *Assignments*: Assign a value to an attribute.
- *Tables*: Describe the relationship among a set of attributes using a matrix representation.

Additionally, CCs also contain non-sequential primitives similar in their semantics to process synchronization primitives like the *join* operation. These kinds of semantics are very hard to express in ordinary flow-charts.

3.2. Design Goals

In order to make KEN accessible to an expert without the need for a human mediator the new user interface should:

1. *enable experts to express their knowledge very efficiently.* Experts usually are very busy persons interested in making the most out of their limited time.
2. *allow users to do simple things simply but also support more complex tasks.* If the interface can only tackle toy-world examples and does not scale up it will be of no use. The interface has to provide mechanisms to deal with complex structures.
3. *be maintainable by the knowledge engineers.* Instead of being a mediator between the expert and the expert system the knowledge engineer should mainly observe frequent transformations from the situation model to the system model required by experts. Then, he should have the ability to incorporate these transformations into the interface.
4. *help users to map their goals to functionality provided.* The old interface provides little guidance for the knowledge acquisition; the first encounter of KEN by an user happens in a empty text-oriented editor window showing a blinking cursor in the upper left corner.
5. *allow the incremental construction of the behavior and the "look" of new functionality.* The specification of new CC primitives should be possible based on existing ones.
6. *support a two-dimensional representation of the problem domain resembling the situation model of users.* The paper-and-pencil representations of experts are typically two-dimensional. The new interface should reflect this fact to narrow the gap between situation model and system model.
7. *provide functionality exceeding the possibilities of existing paper-and-pencil schemes representing situation models.* In order to be motivated to move from one representation to another, experts have to be rewarded by additional functionality that helps them to accomplish tasks in a better way.

The objective of the following sections is to show how these goals were achieved with Agentsheets, what special advantages the Agentsheet solution has, where it is limited, and how these findings can be generalized to other systems.

3.3. Configuration-Charts and Agentsheets

So far, the discrepancy between the situation model and the system model has been discussed. Furthermore, CCs have been identified as an appropriate means to capture the situation model. The following section will elaborate how CCs can be represented by Agentsheets, and how the link from the Agentsheet-based CC representation to the conventional text-based knowledge representation can be accomplished.

Configuration Charts Primitives

Every CC primitive is represented by a single agent. We distinguish between *wire agents* defining the semantics of data propagation, and *control-unit agents* representing typical chunks of procedural expert knowledge like if-then-else conditionals, case statements, hierarchical links to nested CCs, database queries, calls to arbitrary (Lisp) functions, and join operations. Figure 5, below, delineates the corresponding gallery.

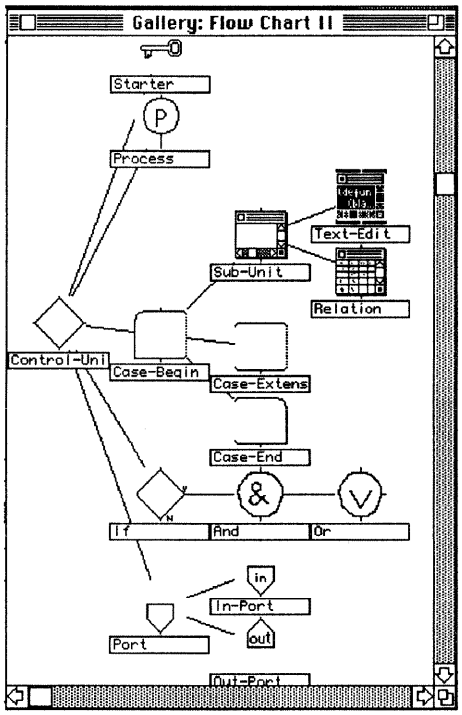


Figure 5: Control Units

Configuring Cars

A simple Car configuration example taken from a one-week KEN course is used as scenario describing interactions between end-users and the Agentsheet-based system. The goal is to create a simple car expert system recommending a car type based on the engine size and the make. Although there are many issues involved (e.g., slot cardinality, slot type, dimensions, bounds, print names, etc.), the conceptual model of the problem can be reduced to a rather trivial decision tree.

A task description of the Car configurator example, and two galleries containing wire agents and control-units agents were given to three users with the goal of specifying the example using the Agentsheet-based CC

interface. Two of the three users have never been exposed to KEN.

By selecting depictions from the galleries and dragging them into an initially empty Agentsheet the users composed a CC like the one pictured in Figure 6. Additionally, every case statement had to be associated with the attribute of the component to be defined employing a option-click sensor of control-units agents. The option-click operation has been explained to the users during the task.

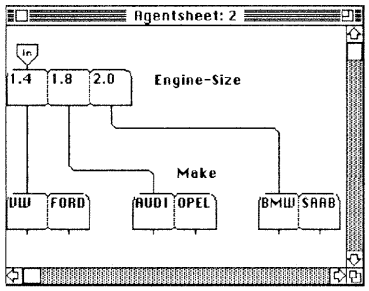


Figure 6: The Car configuration sheet

At the click of a button, the CC generates the established text-based knowledge representation. Even with the simple car example the equivalent frames and demon function definition is certainly less than intuitive to a casual user. The demon functions resulting from our Car example are shown below:

```
(DEMON DERIVE-FROM-ENGINE-SIZE-1
       (TRIGGER-TYPE IF-WAS-ADDED)
       (PLACED (FRAME CAR)
               (SLOT ENGINE-SIZE))
       (NEEDED-SLOTS ENGINE-SIZE)
       (IF (= (FACT 'ENGINE-SIZE :FACET 'BASICFRAMES::VALUE) '1.4))
       (THEN (CONCLUDE 'MAKE 'VW :FACET 'RANGE)
             (CONCLUDE 'MAKE 'FORD :FACET 'RANGE)))

(DEMON DERIVE-FROM-ENGINE-SIZE-2
       (TRIGGER-TYPE IF-WAS-ADDED)
       (PLACED (FRAME CAR)
               (SLOT ENGINE-SIZE))
       (NEEDED-SLOTS ENGINE-SIZE)
       (IF (= (FACT 'ENGINE-SIZE :FACET 'BASICFRAMES::VALUE) '1.8))
       (THEN (CONCLUDE 'MAKE 'AUDI :FACET 'RANGE)
             (CONCLUDE 'MAKE 'OPEL :FACET 'RANGE)))

(DEMON DERIVE-FROM-ENGINE-SIZE-3
       (TRIGGER-TYPE IF-WAS-ADDED)
       (PLACED (FRAME CAR)
               (SLOT ENGINE-SIZE))
       (NEEDED-SLOTS ENGINE-SIZE)
       (IF (= (FACT 'ENGINE-SIZE :FACET 'BASICFRAMES::VALUE) '2.0))
       (THEN (CONCLUDE 'MAKE 'BMW :FACET 'RANGE)
             (CONCLUDE 'MAKE 'SAAB :FACET 'RANGE)))
```

It took all users less than ten minutes to accomplish the task. Taking a KEN course, users were only able to setup the same example, using the lisp-flavored textual representation, after about eight hours training. This comparison is not entirely fair because the course also includes a basic Common Lisp introduction. Nonetheless, the magnitude in difference appears to justify future research.

Does the Agentsheet Approach meet the Design Goals?

We now review the initial design goals and describe how they have been addressed by Agentsheets. Does the Agentsheet-based interface

1. *enable experts to express their knowledge very efficiently.* The Car configurator task revealed a major increase in performance to accomplish a task.
2. *allow users to do simple things simply but also support more complex tasks.* The Car configurator has shown that end-users can define a simple knowledge base with very little effort. Moreover, a hierarchical means of capturing knowledge, provided by the so-called *hyperagent* mechanism of Agentsheets, will probably help to manage more complex systems. Hyperagents allow nesting of CCs. More generally, hyperagents are used to embody abstractions. A hyperagent is a placeholder for an entire Agentsheet. For applications like CCs, hyperagents serve as a hierarchical navigation tool similar to a hypertext system.
3. *support maintenance by the knowledge engineers.* Most extensions to the CC semantics can be addressed on the level of **spatial reasoning** rather than on the level of conventional programming languages (Common Lisp in this case).
4. *help users to map their goals to functionality provided.* The gallery provides a means to visualize potential operations applicable to a task. The gallery serves the function of a **palette** containing concepts priming the expert towards a solution. Some users were distracted by the cloning graph representation of the gallery because the cloning relationships are completely irrelevant to them. This problem has been overcome by copying the depictions from the gallery into an Agentsheet that serves as a palette.
5. *allow the incremental construction of the behavior and the "look" of new functionality.* The object-oriented structure of agents eases the reuse of existing functionality and therefore allows the incremental extension of system built with Agentsheets based on **inheritance**.

   Many new depictions were just variations of existing ones and, therefore, could be **cloned**. For example the wire gallery consisting of 15 different wire depictions (straight, l-shapes, t-shapes, and crossings) could all be cloned from just two basic wire types: a l-shaped wire and a straight wire (see Figure 7). Moreover, the cloning relationships has been used to *derive the*

semantics of the cloned depictions, e.g., a depiction cloned by rotation suggests the creation of a sibling class by "rotating" the methods of the class linked to the source of cloning. The relationship between cloning operation and class creation is, however, not always clear. It will be the subject of future research.

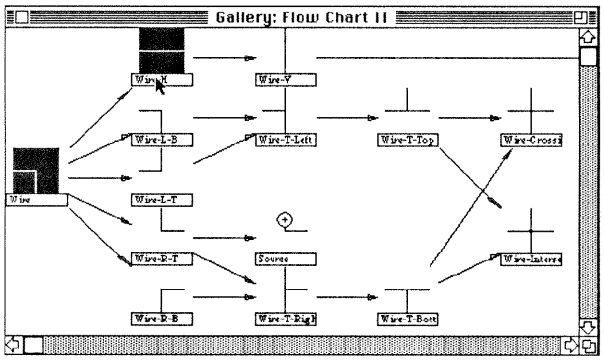


Figure 7: Wire Gallery

6. *support a two-dimensional representation of the problem domain resembling the situation model of users.* Agentsheets are inherently two-dimensional, i.e., elements of paper-and-pencil representations of situation models can be mapped one-to-one to agent depictions.
7. *provide functionality exceeding the possibilities of existing paper-and-pencil representations of situation models.* Agentsheets have inherent direct manipulation [12] properties: dragging of agents, sensors to react to clicking. This interactive nature of agents is employed to provide immediate feed back to the user. In contrast to paper-and-pencil representations, users can operate actively on Agentsheets.

4. Discussion

Agentsheets, in their current stage, are *used* by a casual user, however, they are not ready to be *extended* by the casual user. In the Car configurator case study this turned out to be irrelevant because the designer of new agent classes, the knowledge engineers, were familiar with object-oriented programming. This is of course not generally true, i.e., many users have not been exposed to object-oriented programming. Also, McLean points out that concepts of object-oriented programming can be very hard to grasp for non-programmers [8]. Moreover, most users just want to get their job done. That is, they do not see any immediate reward in learning yet another programming language.

Although the style of programming on the spatial reasoning level of Agentsheets can be compared in its simplicity with formulas of spreadsheets, it still reflects some aspects of the underlying Common Lisp layer. Our current research is concerned with the replacement of this procedural programming style with a more declarative one. One idea is to attach Prolog predicates having a two-dimensional argument structure [17] to agents.

5. Conclusions

Object-oriented techniques have been used for a while to define the behavior of objects. However, the use of a gallery representing cloning relationships among depictions of objects, and the fact that these cloning relationships can be used to derive new behavior make Agentsheets a new, unique approach to design and implement user interfaces.

In the hands of occasional programmers, or knowledge engineers in our case study, Agentsheets have shown to be an efficient means for creating and extending intelligible applications. The ability to define the behavior and the look of agents in terms of existing agents empowers users to construct the specification of their applications incrementally [2].

Acknowledgements

This research was funded by the United States Bureau of Reclamation through the Advanced Decision Support program. We wish to express our thanks to Clayton Lewis and Brent Reeves who provided valuable feedback for the Agentsheet prototype. Roland Hübscher was a source of many enlightening discussions. Michael Vitins furnished very helpful information concerning the configuration domain. ABB CADE AG graciously funded the Configuration Charts case study project.

References

[1] G. Cattaneo, A. Guercio, S. Levialdi and G. Tortora, "ICONLISP: An Example of Visual a Programming Language," *IEEE Computer Society, Workshop on Visual Languages*, Dallas, 1986, pp. 22-25.

[2] G. Fischer, "Communication Requirements for Cooperative Problem Solving Systems," *Informations Systems*, Vol. 15, pp. 21-36, 1990.

[3] M. R. Genesereth and N. J. Nilson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufman Publishers, Inc., Los Altos, 1987.

[4] E. P. Glinert, "Towards "Second Generation" Interactive, Graphical Programming Environments," *IEEE Computer Society, Workshop on Visual Languages*, Dallas, 1986, pp. 61-70.

[5] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1983.

[6] D. Ingalls, S. Wallace, Y.-Y. Chow, F. Ludolph and K. Doyle, "Fabrik: A Visual Programming Environment," *OOPSLA '88*, San Diego, CA, 1988, pp. 176-190.

[7] C. Lewis, "NoPumpG: Creating Interactive Graphics with Spreadsheet Machinery:," *Technical Report*, CU-CS-372-87, Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0430, 1987.

[8] A. MacLean, K. Carter, L. Lövstrand and T. Moran, "User-Tailorable Systems: Pressing the Issues with Buttons," *CHI '90*, Seattle, WA., 1990, pp. 175-182.

[9] M. Minsky, "A Framework for Representing Knowledge," in *The Psychology of Computer Vision*, P. H. Winston, Ed., Mc Craw-Hill computer science series, 1975, pp. 211-277.

[10] K. W. Piersol, "Object Oriented Spreadsheets: The Analytic Spreadsheet Package," *OOPSLA '86*, 1986, pp. 385-390.

[11] M. Rettig, T. Morgan, J. Jacobs and D. Wimberley, "Object-Oriented Programming in AI," *AI Expert*, , pp. 53-69, 1989.

[12] B. Schneiderman, "Direct Manipulation: A Step Beyond Programming Languages," in *Human-Computer Interaction: A multidisciplinary approach*, R. M. Baecker and W. A. S. Buxton, Eds., Morgan Kaufmann Publishers, INC.95 First Street, Los Altos, CA 94022, Toronto, 1989, pp. 461-467.

[13] N. C. Shu, "Visual Programming: Perspectives and Approaches," *IBM Systems Journal*, Vol. 28, pp. 525-547, 1989.

[14] M. Vitins, "KEN - A Frame-based Expert System Shell," *Technical Report*, TN CRBC 02/87, ABB Asea Brown Boveri Research Center, 1987.

[15] M. Vitins, *The KEN User Manual*, ABB Press, Baden, 1989.

[16] N. Wilde and C. Lewis, "Spreadsheet-Based Interactive Graphics: From Prototype to Tool," *Technical Report*, CU-CS-445-89, University of Colorado at Boulder, Department of Computer Science, Campus Box 430, Boulder, Colorado 80309-430, 1989.

[17] R. Yeung, "MPL - A Graphical Programming Environment for Matrix Processing Based on Logic and Constraints," *IEEE Workshop on Visual Languages*, Pittsburg, 1988, pp. 137-143.

[18] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka and T. Ichikawa, "Interactive Iconic Programming Facility in HI-VISUAL," *IEEE Computer Society, Workshop on Visual Languages*, Dallas, 1986, pp. 34-41.