A SPREADSHEET INTERFACE FOR LOGIC PROGRAMMING

Michael Spenke and *Christian Beilken*

Gesellschaft für Mathematik und Datenverarbeitung mbH
(German National Research Center for Computer Science)
P.O. Box 1240
D-5205 Sankt Augustin 1
Federal Republic of Germany
Email: spenke@gmdzi.uucp and cici@gmdzi.uucp

ABSTRACT

We present PERPLEX, a programming environment intended for the end-user. In its design, the concepts of logic programming and spreadsheets are combined. Thus, on the one hand, logic programming becomes an interactive, incremental task where the user gets direct visual feedback, on the other hand, functionality and scope of a conventional spreadsheet program are considerably extended. In order to perform calculations and queries, constraints are imposed on the contents of the spreadsheet cells. New predicates can be defined using a programming-by-example technique: Rules are extracted from the user's solutions for example problems. Thus, concrete intermediate results take over the role of abstract logic variables in the programming process. PERPLEX has been successfully implemented on a Symbolics Lisp Machine.

KEYWORDS: End-user programming, programming by example, logic programming, graphical user-interface, constraints, spreadsheet, database queries.

INTRODUCTION

Highly interactive, graphical user-interfaces have considerably simplified the use of standard application software packages. However, there is also a great demand for more specialized applications, tailored to the needs of individual users [21]. This demand is still unsatisfied, and there is no hope of satisfying it by an increased number of professional programmers [22]. Therefore, tools are needed which allow end-users to develop their own specialized applications.

PERPLEX combines the power of logic programming with the popular user interface of spreadsheets. Incremental queries are introduced as a natural extension of the standard functionality of a spreadsheet. A *programming-by-example technique* is used to create user-defined functions and logic rules in a uniform manner without the need for a new formal language.

One reason for the tremendous success of *spreadsheet programs* is that the matrix format leads to a natural representation of many problems. Even more important, the traditional distinction between program design and program testing is abolished, so that programming becomes a less abstract task: The current values of the variables are permanently displayed. In many cases errors are immediately detected and an explorative style of programming is encouraged, because the user can quickly see the effects of changes to input values or formulas. Instead of typing variable names and trying to imagine their expected values, the user can point at concrete intermediate results.

However, a conventional spreadsheet program does not constitute a complete programming environment, mainly because it does not incorporate user-defined functions or procedures in order to structure large programs [1]. Macros are an attempt to overcome this deficiency but they are a step back to traditional programming languages. Therefore, macros are a great barrier for the average spreadsheet user. While many people use spreadsheets, there are only a few who use the macro extensions. In PERPLEX, programming is as easy as using a spreadsheet

Logic programming languages (like Prolog) promise to be more problem oriented and easier to learn for the non-programmer than procedural languages. Furthermore, they are more expressive than functional languages because of their flexible input-output directions [19], and can be used as database query languages in a straightforward way [4].

Logic programming languages are designed to be used *interactively.* This large potential is left unexploited as long as a simple line-oriented interface is used. The truely interactive, graphical interface of PERPLEX makes the concepts of logic programming available to end-users.

Van Emden et al. [9] also use a matrix display to present answer substitutions of incremental Prolog queries. However, they do not support the definition of new predicates. Kriwaczeck [12] uses a spreadsheet only to display the contents of a database of assertions.

In PERPLEX, database queries are expressed in a quite similar way as in the Query-by-Example language [26]. However, database queries are only a special case of our more general concept.

BASIC CONCEPTS

At first glance, PERPLEX looks like a traditional *spreadsheet program*: The display consists of cells organized in rows and columns. Individual cells represent *variables* which are named A1, A2, ..., B1, B2, ... Each cell displays the current value of its variable, if any. Constants can be numbers, strings, or lists of constants.

Constraints versus Functional Expressions

In a typical spreadsheet program, a cell can either hold a constant or a formula. Cells are thus strictly divided into *input cells* (those containing a constant) and *output cells* (computed by formulas). In PERPLEX, however, formulas are not attached to a single cell, but apply to a *set of cells*. Consequently, formulas do not have the form of functional expressions which return a value, but are given as *constraints*, which have to be satisfied by the contents of the cells. Examples for constraints are:

- The value of A1 must be twice the value of B1.
- The value of A1 must be greater than the value of B1.
- The sum of A1, B1, and C1 must equal 100.
- The person named in A1 must be the manager of the person in B1.

The list of constraints entered by the user is displayed beside the spreadsheet. While the formulas of a conventional spreadsheet are specified using *functions*, constraints are specified using *predicates*, which do not imply a fixed input-output direction. The basic syntax of a constraint is **P(x_1 .. x_n)**, where **P** is a predicate and x_1 .. x_n are cell references or constants. The constraint list is analogous to a Prolog query.

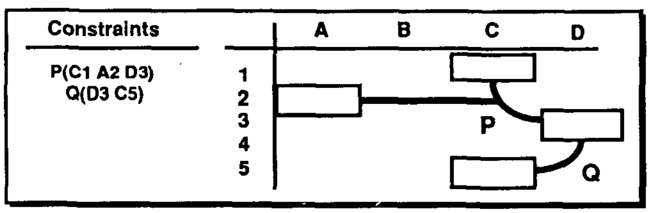

Figure 1: A sheet with two constraints

Constraint Satisfaction

The task of the evaluator is to find a *substitution* for all variables referenced in the constraint list (which do not have a constant value), such that all constraints are satisfied. If a substitution is found, the value for each variable is displayed in the spreadsheet. Each substitution constitutes a solution for the query expressed by the constraints. In general, there can be more than one solution, which will be shown one after the other by scrolling. While scrolling, the values displayed for all variables change simultaneously, because each substitution is a global solution for all constraints. It is also possible that no solution is found at all.

Other than in a conventional spreadsheet, it is left open whether cells are designated as input- or output-cells. Consider the first example constraint:

- If A1 and B1 are both given as constants, the evaluator will just check whether the constraint is satisfied.
- If only B1 is given, A1 will be computed by multiplying B1 by 2.
- If only A1 is given, B1 will be computed by dividing A1 by 2.
- If neither A1 nor B1 are given, the constraint is ignored by the evaluator.

With each predicate, its set of *legal input-output modes* is stored. The modes define which parameters have to be given when the predicate is called. Formally, each mode is a subset of the formal parameters of a predicate. If all parameters of at least one legal mode are supplied, the constraint becomes *evaluable*, i.e. the values of the remaining parameters can be computed. The mode information is used to determine the *order of evaluation* of constraints: It is always the first evaluable constraint, which is evaluated next. Evaluation of a constraint may instantiate further variables, so that more constraints become evaluable. When no further constraint is evaluable, the remaining constraints are ignored by the evaluator.

A legal mode of a predicate is written as a *regular expression* of the symbols **in** and **out**. For example, **(in in out)**, as a legal mode of the predicate **Times**, means that the product (the last parameter) can be computed if both factors (the first two parameters) are given. It is not necessary to specify all legal modes of a predicate explicitly, because supplying constants for out-parameters is always allowed.

Different Kinds of Predicates

Table 1 lists some typical predicates and their legal input-output modes.

Predicate	Legal Modes	Example
Less	(in in)	Less(A4 5)
Times	(in in out)	Times(A3 2 A4)
	(in out in)	
	(out in in)	
And	(in in)	And(true true)
Member	(out in)	Member(A1 [1 2 3 4 5])
List	(in* out)	List(1 2 3 4 5 B1)
	(out* in)	List(A1 A2 A3 [1 2 3])
Concat	(in in out)	Concat("in" "put" A3)
	(out out in)	Concat(A1 A2 "output")
Substring	(in out out out)	Substring("core" 2 3 B4)
Employees	(out out out out)	Employees(A1 A2 A3 A4)
Copy File	(in in)	Copy File("file1" "file2")
Pie Chart	(in in in)	Pie Chart("Sales" A2 A4)

Table 1: Some typical predicates

There are *three different types* of predicates which are all used in the same way, but differ in their underlying implementation:

- **Built-in predicates**
  A built-in predicate is implemented by a set of functions — one for each legal mode. Each function has in-parameters as indicated in the mode-expression and returns a list of solutions for the out-parameters. For example, for the mode **(in in out)** of **Times**, there is a function that gets two numbers and returns the product of the numbers.

- **Database relations**
  Database relations are represented by a set of tuples. They are similar to lists of facts in Prolog.
- **User-defined predicates**
  User-defined predicates can be defined using built-in predicates, database relations, and other user-defined predicates. This is discussed below.

Backward Calculations

Figure 2 shows how a simple computation can be specified by constraints.

Constraints		A	B	C
Plus(B2 100 C2)	1	Amount	200	
Divide(C2 100 C3)	2	Percent	14	114
Times(B1 C3 B3)	3	Total	228	1.14
	4			

Figure 2: Adding a percentage using explicit intermediate results

Once we have defined these three constraints, we can not only experiment with different input values, but also do *backward computations*. For example, if we supply the basic amount (cell B1) and the desired result (B3), the percentage will be computed. In order to get the result, the evaluator has to find an appropriate order of evaluation for the three constraints. As mentioned above, it is always the *first evaluable constraint* that will be chosen. In this case, the third constraint can be evaluated first. As a result, the second becomes evaluable and finally the first constraint is chosen.

Syntactic Sugar

Since it is quite unusual to write down arithmetic formulas in predicate-notation, the more convenient standard infix-notation is also offered for constraints. For example, instead of the three constraints in Figure 2, we can write the shorter:

B3 = B1 * (B2 + 100) / 100

Nevertheless, the internal representation is still based on constraints (with auxiliary variables). Since the evaluator works on the basis of the constraint representation, backward computation is still possible.

It is also possible to call predicates like functions within *nested expressions*. The last parameter of the predicate must then be left out and is delivered as the result of the function call. For example,

A3 = Sinus(A1) + A2

is equivalent to

Sinus(A1 AUX1)
Plus(AUX1 A2 AUX2)
Equal(AUX2 A3)

Furthermore, operators like **>**, **<**, **and**, **or**, and **not** can be used to construct either Boolean expressions (returning **true** or **false**) or constraints (which have to be satisfied). For example,

A1>5 and A2<10 and not A3

will fail, if one of the three conditions is not satisfied, while

A4 = (A1>5 and A2<10 and not A3)

will assign either **true** or **false** to A4.

Multiple Solutions

Typically, multiple solutions are obtained when database relations are involved. Database queries can be easily formulated using the same basic mechanisms as described so far. In our examples, a small relational database of 10 employees (introduced in [26]) is used (see Table 2).

Name	Salary	Manager	Department
anderson	6000	murphy	toy
henry	9000	smith	toy
hoffman	16000	morgan	cosmetics
jones	8000	smith	household
lewis	12000	long	stationery
long	7000	morgan	cosmetics
morgan	10000	lee	cosmetics
murphy	8000	smith	household
nelson	6000	murphy	toy
smith	12000	hoffman	stationery

Table 2: The **Employees** database

When **Employees(B1 B2 B3 B4)** is entered, the evaluator finds all tuples stored in the database as solutions. The data of the first employee are displayed in cells B1 .. B4 and the total number of solutions is shown above the sheet. Other solutions are shown, when one of the two scroll-arrows is pressed.

-- Solution 1 of 10 --

Constraints		A	B	C
Employees(B1 B2 B3 B4)	1	Name:	anderson	
	2	Salary:	6000	
	3	Manager:	murphy	
	4	Dept:	toy	
	5			

Figure 3: A simple database query

More specific queries can be asked by supplying constants for some of the fields of the database. For example, if we enter **jones** into B1 there is a unique solution: Only the data of Jones are displayed. Alternatively, if we enter **6000** into B2, 2 solutions will remain, since there are 2 employees who earn $6000.

Combined Database Queries

For more complicated queries, additional constraints can be specified. For example, to ask for all employees earning more than $8000, we add the constraint **B2 > 8000** and get 5 solutions.

Instead of adding further restrictions on the database fields, we can also compute information derived from the displayed results: If we want to know the annual salary of each employee, we add the constraint **C2=B2*12**.

It is also possible to combine two database look-ups in a single query: Suppose we want to know which employees have the same manager. Again we start with **Employees(B1 B2 B3 B4)** and enter **jones** into B1 as an example, so that we have just one solution and find Smith as the manager of Jones. Now we want to know which employees also have Smith as their manager. We add the constraint **Employees(C1 C2 C3 C4)** for the next column and also **B3 = C3**. We get 3 solutions and the colleagues of Jones are displayed in cell C1. Scrolling through the solutions we find that Jones is among the colleagues and our query is not yet perfect. So we add another constraint, namely **B1 ≠ C1**, and only 2 colleagues remain. Finally, we can clear the entry **jones** in B1 and get 10 pairs of employees, having the same manager.

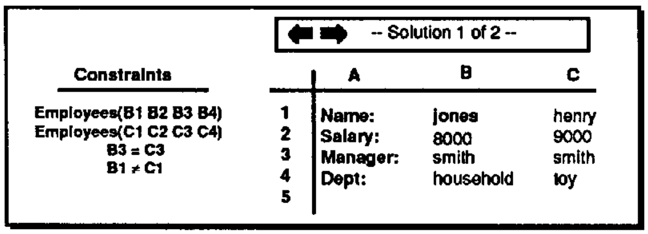

Figure 4: "Who has the same manager as Jones?"

Incremental Problem Solving

As the last example has shown, the strength of PERPLEX lies in the *interactive, explorative style* it supports in attacking a problem. The user is not forced to specify a complete query from scratch. Instead, after each step, intermediate results are immediately displayed. Depending on the results obtained, further constraints can be added to exclude unwanted solutions or existing constraints can be modified or removed if they are too restrictive. The specification of additional constraints is easy because the user does not have to deal with abstract variables and imagine their meaning, but instead can point at intermediate results and say: "This should be less than that".

PROGRAMMING BY EXAMPLE

In the examples discussed so far, it was demonstrated how existing predicates can be used for incremental problem solving. New user-defined predicates are defined by a *programming-by-example* technique.

The basic idea of programming by example is to first tackle an *example problem* in a trial and error fashion, and in case of success, identify the relevant parameters of the problem. The user starts by entering some constants describing the example problem. Then he defines constraints, obtains intermediate results, modifies constraints and so on. Eventually, the desired results will appear in the sheet. At this time, a certain relation has been established between the input cells, where initially the example constants were entered, and the output cells, where the relevant results are displayed. The relation is defined by the constraints connecting the cells. The user can now define a new user-defined predicate by assigning a name to the relation between input and output cells, thus stating that the same relation shall hold for the parameters of the new predicate.

It is possible to give several examples for one user-defined predicate resulting in a separate rule for each example. Thus, **recursive** predicates can also be defined.

An Example for an Example

Suppose we are repeatedly faced with the problem of substituting a part of a word by another string. As an example problem, we try to substitute "act" in "interaction" by "sect". Our basic idea is to cut the word "interaction" into the prefix "inter", the infix "act" and the postfix "ion" and then to concatenate the prefix, the new infix and the postfix.

First we enter the example constants into our sheet and add some appropriate labels indicating the meaning of the input fields (Figure 5). Then we use the predicate **Substring** to find out the start and end position of "act" in "interaction". The positions 6 and 8 are displayed as the result. Next we try to build the prefix as the substring of "interaction" from 1 to 6. As soon as we see the result, "intera", we know that we have made a mistake: The start position of "act" has to be decremented by one in order to get the end position of "inter". So we have to modify the last constraint slightly. To compute the postfix "ion", we first need the length of the word "interaction". Therefore, we add another constraint using the predicate **String-length**. Using the result, we can determine the postfix "ion" by another **Substring** constraint. Finally, we concatenate the three parts.

As we are satisfied with the solution, we want to define a new predicate **Substitute** with the parameters **String**, **Substring**, **Substitute**, **Result** such that **Substitute("interaction" "act" "sect" "intersection")** holds. Therefore, we select the **Define Rule** menu command, enter the name of the new predicate and click at the relevant parameter cells in the correct order. We have stated that the parameters of the newly defined predicate **Substitute** shall be in the same relation as the cells we have pointed at.

The new predicate can be used to solve substitution problems like the original example: The parameters **String**, **Substring** and **Substitute** are supplied and **Result** is computed. However, there are many other ways to use the predicate **Substitute**, because of its legal input-output modes:

- **Substitute("interaction" "i" "o" A4)** will find the two solutions "onteraction" and "interactoon".
- **Substitute("interaction" A2 "*" A4)** will substitute each of the 78 possible substrings of "interaction" by "*".
- **Substitute("interaction" A2 "sect" "intersection")** will find out that "act" must be replaced.
- **Substitute("interaction" A2 A3 "intersection")** will find all 36 possible replacements required to come from "interaction" to "intersection".

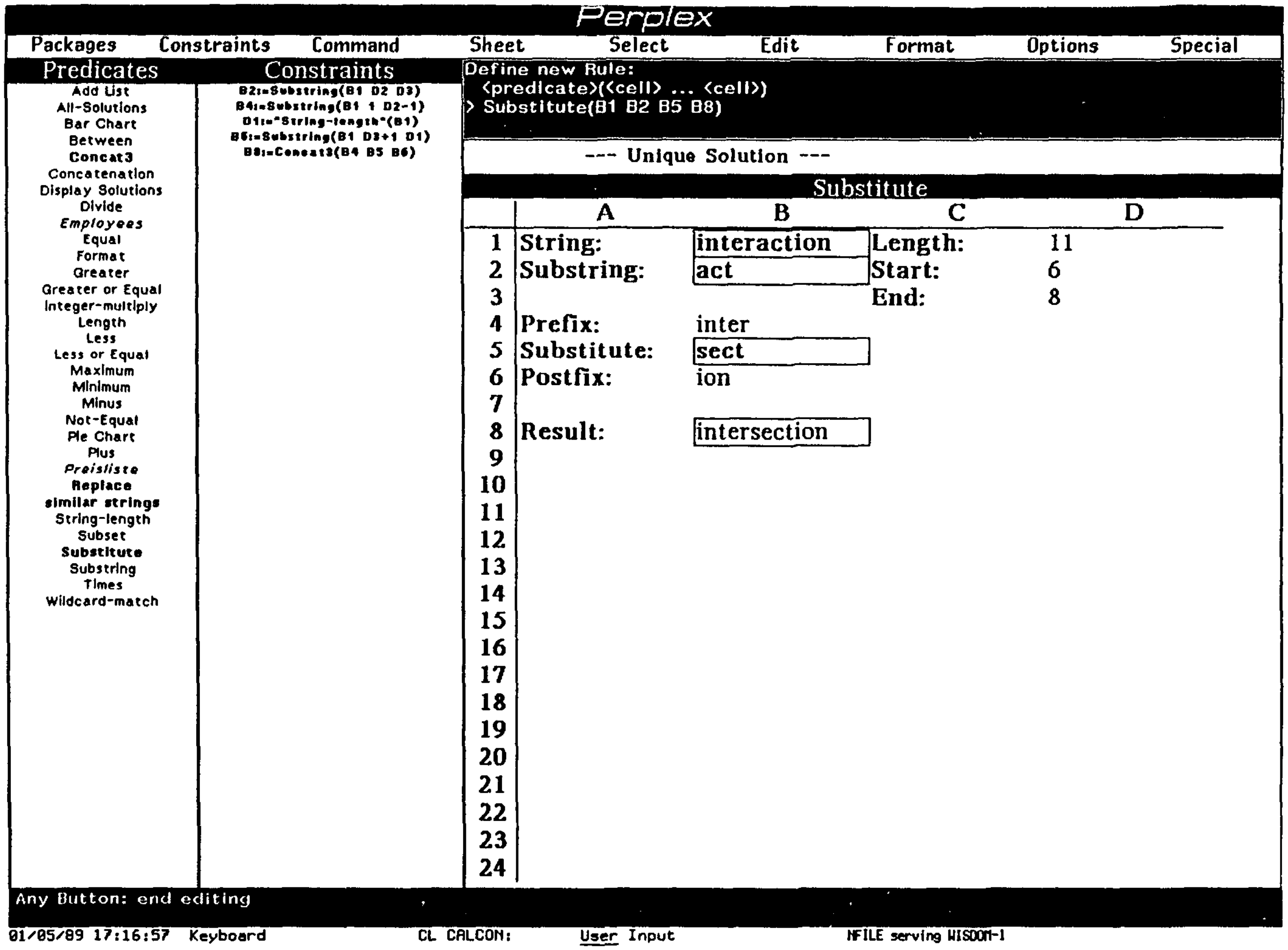

Figure 5: Exemplary solution of the substitution problem (screen dump)

Advantages of Programming by Example

The above example shows some important advantages of our programming by example approach:

- There is no new formalism or language the user has to learn in order to define new predicates. Programming general solutions is almost as easy as solving a single, concrete problem. The user need not even know in advance that he is writing a program. Only when a problem is solved that seems to be of general importance and is expected to occur again, the user can decide to define a new predicate and identify the relevant parameters. Thus, the explorative style of working can also be used to define user-defined predicates.
- The legal input-output modes of new predicates are *automatically* computed. The user can immediately verify whether his exemplary solution could be *generalized* to the desired extent. Often, it is possible to work with a very simple example with more parameters supplied than necessary, resulting in unique intermediate and final results and yet implement the general case where less parameters are supplied and multiple solutions are possible. It is also possible that a forward computation is demonstrated, and the system detects the possibility to perform backward computations with the new predicate. Thus, logic programming appears as a natural extension of functional programming.
- Since a rule is represented as the sheet containing the original example, there is a natural way to *edit* existing rules: The defining sheet of a rule can be opened and modified with the usual operations.

CONCLUSION

We have shown that it is possible to combine the power of logic programming concepts with the popular user interface of spreadsheets in a natural way. The explorative style of programming encouraged by the direct visual feedback of spreadsheets can also be used to perform incremental queries, and even to define user-defined predicates by a programming-by-example technique.

PERPLEX has been successfully implemented on a Symbolics Lisp Machine [23]. First results of an interaction analysis [10] comparing PERPLEX and EXCEL [6] indicate that our programming paradigm is convenient for the end-user, but users have some problems with the user interface of the Lisp Machine. Currently, we are working on a reimplementation for the Macintosh.

REFERENCES

1. Deane E. Arganbright: **Mathematical Modeling with Spreadsheets,** ABACUS, 3(4) Summer 86, pp 18-31.
2. Giuseppe Attardi, Maria Simi: **Extending the Power of Programming by Example,** in: J.O. Limb (Ed.): SIGOA Conference on Office Information Systems, Philadelphia, PA, 21-23 June 1982, pp 52-66.
3. Michael A. Bauer: **Programming by Examples,** Artificial Intelligence 12,1979, pp 1-21.
4. Rudolf Beyer: **Database Technology for Expert Systems,** in: Internationaler GI-Kongreß 85: "Wissensbasierte Systeme", 29. October 1985, München, GI Tagungsband, Informatik Fachberichte 112, Springer-Verlag 1985, pp 1-16.
5. Alan Borning: **The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory,** ACM TOPLAS, 3(4), October 1981, pp 353-387.
6. Mary V. Campbell: **Excel Macro Treasury,** Macworld, November 1987, pp 122-125.
7. Alain Colmerauer: **Opening the Prolog III Universe,** BYTE, August 1987, pp 177-158.
8. Marc Eisenstadt, Tony Hasemer, Frank Kriwaczek: **An Improved User Interface for PROLOG,** In: B. Shackel (Ed.): Human-Computer Interaction—INTERACT '84, Elsevier Science Publishers B.V., 1985, pp 385-389.
9. M.H. van Emden, M. Ohki, A. Takeuchi: **Spreadsheets with Incremental Queries as a User Interface for Logic Programming,** ICOT Technical Report, TR-144, October 1985.
10. Gernoth G. Grunst: **Reconstructing Cooperative Advising—Interaction Analysis as a Research Tool for the Development of Intelligent Assistants,** internal Paper, 1988, GMD.
11. Daniel C. Halbert: **Programming by Example,** PhD Thesis, Computer Science Division, Dept. of EE&CS, University of California, Berkeley 1984.
12. Frank Kriwaczek: **LogiCalc—A PROLOG Spreadsheet,** in: Bob Kowalski, Frank Kriwaczek: Logic Programming, Addison-Wesley 86, pp 105-117.
13. Catherine Lassez: **Constraint Logic Programming,** BYTE, August 1987, pp 171-176.
14. Wm Leler: **Constraint Programming Languages, Their Specification and Generation,** Addison-Wesley 1988.
15. Henry Lieberman, Carl Hewitt: **A Session With TINKER: Interleaving Program Testing With Program Design,** Conference Record of the 1980 LISP Conference, Stanford University, August 1980, pp 90-99.
16. Brad A. Myers: **Visual Programming, Programming by Example, and Program Visualization: A Taxonomy,** in: Marilyn Mantei, Peter Orbeton (Eds): Human Factors in Computing Systems—III, Proceedings of the CHI '86 Conference, Boston, MA, 13-17 April 1986, pp 59-66.
17. M. Ohki, A. Takeuchi, K. Furukawa: **A Framework for Interactive Problem Solving based on Interactive Query Revision,** ICOT Technical Report, TR-188, June 1986.
18. Kurt W. Piersol: **Object oriented Spreadsheets: The Analytic Spreadsheet Package,** ACM OOPSLA Proceedings, September 1986, pp 385-390.
19. Uday S. Reddy: **On the Relationship between Logic and Functional Languages,** in: Doug De Groot, Gary Lindstrom (Eds): Logic Programming, Functions, Relations and Equations, Prentice Hall, 1986, pp 3-35.
20. Robert V. Rubin, Eric J. Golin, Steven P. Reiss: **ThinkPad: A Graphical System for Programming by Demonstration,** IEEE Software, March 1985, pp 73-79.
21. John F. Rockart, Lauren S. Flannery: **The Management of End User Computing,** Communications of the ACM, 26(10), October 1983, pp 776-784.
22. Nan C. Shu: **FORMAL: A Forms-Oriented, Visual-Directed Application Development System,** IEEE Computer, August 1985, pp 38-49.
23. Michael Spenke, Christian Beilken: **The Implementation of PERPLEX: A Spreadsheet Interface for Logic Programming by Example,** Internal Research Report FB-GMD-88-29, GMD, 1988, 86 pages.
24. Gerald Jay Sussman, Guy Lewis Steele Jr.: **CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions,** Artificial Intelligence 14, 1980, pp 1-39.
25. Moshé M. Zloof, S. Peter de Jong: **The System for Business Automation (SBA): Programming Language,** Communications of the ACM 20(6), June 1977, pp 385-396.
26. Moshé M. Zloof: **Query-by-Example: a data base language,** IBM System Journal 16(4), 1977, pp 324-343.
27. Moshé M. Zloof: **QBE/OBE: A Language for Office and Business Automation,** IEEE Computer 14(5), May 1981, pp 13-22.

ACKNOWLEDGEMENT

This work was conducted as part of the joint project WISDOM and was supported by the German Federal Ministry of Research and Technology and Triumph-Adler AG, Nürnberg, under grant no. ITW8404B.