# Developing constraint-based applications with spreadsheets

A. Felfernig, G. Friedrich, D. Jannach, C. Russ, and M. Zanker

Computer Science and Manufacturing
Universität Klagenfurt
A-9020 Klagenfurt, Austria
{felfernig,friedrich,jannach,russ,zanker}@ifit.uni-klu.ac.at

**Abstract.** Spreadsheets are in wide-spread industrial use for light-weight business applications, whereby the broad acceptance is both founded on the underlying intuitive interaction style with immediate feedback and a "programming model" comprehensible for non-programmers. In this paper we show how the spreadsheet development paradigm can be extended to model and solve a special class of search and optimization problems that occur in many application domains and would otherwise require the involvement of specialized knowledge engineers.

## 1 Introduction

Spreadsheet applications are in wide-spread industrial use for supporting several business processes like, e.g., back-office data analysis or cost estimation and quotation in the sales process. The broad acceptance of these systems is founded on two main pillars: First, the intuitive (tabular) spreadsheet user interface paradigm allows the user to directly manipulate the data. In addition, immediate user feedback is given in a way that the user is capable of detecting the effects of inputs and changes such that different scenarios can be easily analyzed. In fact, because many users are already accustomed to this specific interaction style, a spreadsheet-like interface can be used as mere front-end for more complex applications that would otherwise require more technical knowledge by the domain-expert [10, 12]. On the other hand, from the software development point of view, spreadsheets are a valuable tool that allow non-programmers to develop their own, small applications that support them in their work without requiring classical programming skills: The spreadsheet development model does (in principle) not require that the user understands concepts like *variables* or *program flow* in the first place. Moreover, the test and debugging process is alleviated for these applications as the user can immediately see the effects of changes in his/her program, i.e., the formulae. However, the main restriction of pure spreadsheet applications is their limited functionality, i.e., the main concepts are cells containing values and a set of predefined (mostly arithmetic) functions that contain references to other cells or areas. For these cases where complex business logic is required, industrial-strength spreadsheets environments like Microsoft Excel offer the possibility to use an imperative (scripting) language for extending the application's functionality. Such extensions, however, require classical programming skills. Thus, the idea of having the domain expert as software engineer who develops his small software system based on his requirements is no longer possible.

Although classical Knowledge-based Systems (KBS) support the strict separation from processing logic and domain knowledge that is encoded as rules or constraints in a declarative way, there are several obstacles that prohibit non-programmers from encoding this knowledge without a knowledge engineer: First, existing systems have specific underlying conceptual models, e.g., some sort of Logic Programming, which is in many cases hard to comprehend even for programmers and require a paradigmatic shift in the way people are thinking of software systems and the problems to be solved. In addition, in many cases a problem arises also in terms of notation and syntax, i.e., given a certain lack of high-level modeling tools, the knowledge has to be encoded in a tedious, non-intuitive notation. Finally, these tools are often implemented using more or less academic languages and programming environments which makes the integration into the company's surrounding software infrastructure problematic.

Within this paper we present an approach to develop a special class of business application systems based on the extension of the functionality of standard spreadsheet environments; our goal is to narrow the gap between classical KBS development and standard software engineering processes using a interface that smoothly integrates the advanced reasoning capabilities into the spreadsheet interface paradigm. Our work is driven by real-world requirements and application scenario from the domain of intelligent product configuration outlined in Section 2. In Section 3 we describe the architecture of the CsSolver (Constraint Satisfaction Solver) system consisting of a domain-independent object-oriented constraint solver and its integration into the *Microsoft Excel* spreadsheet system. In the final sections, we discuss related work in the field and give our conclusions based on first evaluations of the prototype system.

## 2 Application domain / Example problem

Product configuration[1] deals with tailoring a configurable artifact (product or service) according to specific customer needs. The market demand for such systems is driven by the business model of *mass customization* [8] : nowadays, the variety of customizable goods and services ranges from personal computers, cars, or clothing to complex technical systems like telecommunication switches [4].

The result of an interactive configuration process is a customer-specific variant of the product, whereby this product constellation conforms all given business rules or technical constraints and is good or optimal with respect to some objective function. Over the last decades, several AI-techniques have been successfully applied to configuration problems [13]. In particular, Constraint Satisfaction techniques [14] have shown to be specifically suited for these problems due to the simple, declarative knowledge representation paradigm and efficient solving algorithms. Informally speaking, a Constraint Satisfaction Problem (CSP) consists of a set of problem variables, each one assigned a finite domain of atomic values, and a set of constraints describing legal combinations of value assignments to the variables. A solution to a CSP is a value assignment to each problem variable such that no constraint is violated.

Let us consider a simplified example that shows the correspondence of CSPs to configuration problems from the domain of sales configuration of private

[1] Note, that product or sales configuration must not be confused with Software Configuration Management.

telecommunication switches. In this business scenario, the sales engineer together with the customer tailor the telecommunication switch by selecting values from a set of features like *max. number of subscriber lines* or *Voice-over-IP* support.

In the CSP representation, the variables with their domains correspond to the different options, e.g.,

$OperatingSystem = \{NT4.0, W2000\}, BasicModel = \{T200, T300, T330\}$,
$voiceIP = \{yes, no\}, price = \{20000...50000\}$
$Manuals = \{English, German, French, Italian\}$

The example constraints in our example are:

*The T300 is always delivered with voiceIP option.*
*No voiceIP available for the T200 model.*
*Manuals for T330 are English and German only.*

Let us further assume that the price is determined by some function based on the individual user choices and other pricing rules. Amongst others, the key requirements of an actual implementation are typically as follows:

- the user should not be forced to specify values for all choices, i.e., the system should be able to compute fitting values,
- the system should give immediate feedback on incompatible choices or remove choices that are no longer available due to prior selections,
- the system should be able to compute a good/optimal solution with respect to some objective function.

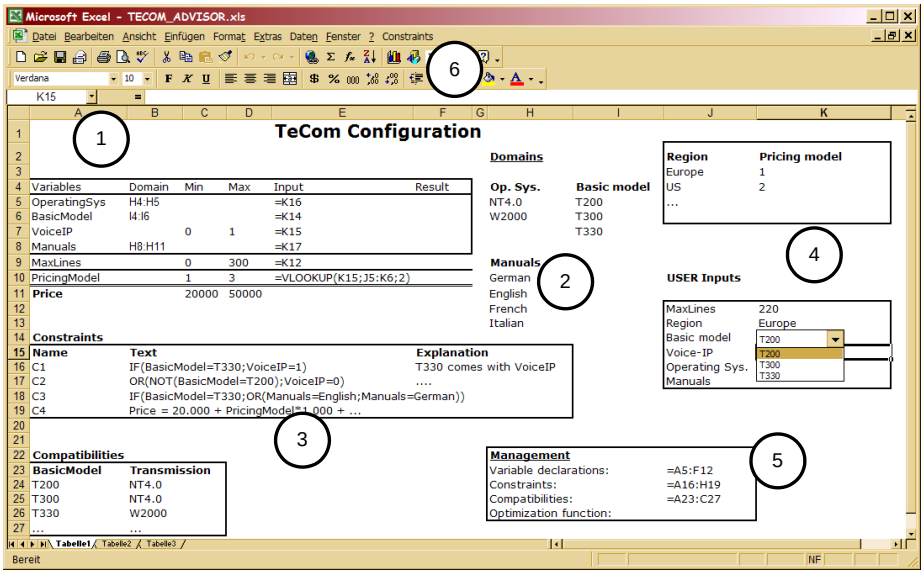


**Fig. 1.** Application screenshot for telecommunication switch configuration

The mapping of this simplified application problem to a CSP is quite straightforward. However, due to the shortcomings of currently available systems the development of such an application typically requires significant coding efforts and highly-specialized development staff. Moreover, the transformation of the domain knowledge to the expert system's representation requires a possibly error-prone knowledge engineering process.

In order to overcome these problems, we developed the CsSolver system as an extension to standard spreadsheets environments; the major design goals were to fit our extensions to the interaction and development paradigm for spreadsheets as much as possible while hiding the technical details of the underlying reasoning process. Figure 1 shows, how the example problem described above can be solved using our system.

In the first step (1) the user defines the named problem variables with their domain. This domain can either be an integer interval or an enumeration of symbolic values in a specified area (2). In addition, the developer can specify what the (user) inputs for the problem variables are, by referencing to other parts of the spreadsheet (4). Note that all the standard functionality of the spreadsheet system can be used for computing the input from e.g., some static data from the application domain. In our case, for instance, the input for the field "pricing model" is computed by looking up the pricing model in (4) that corresponds to the input from K13 (the customer's geographic location). The placement and layout of the fields for the user input can be arbitrarily chosen by the developer. In addition, within modern spreadsheet environments, it is also possible to integrate external data sources smoothly into the application, for instance via an ODBC (Open Database Connectivity) interface. This is a key requirement in our application domain of product configuration where product data is already stored in an e.g., underlying Enterprise Resource Planning system; moreover the results of the configuration process have to be transferred back to this system for order generation and invocation of other business processes.

In the next step (3) the developer defines the problem constraints: This can be done in two ways. In many domains, compatible value tuples for individual variables are naturally given in compatibility tables, i.e., the constraint is defined in *explicit* form by enumerating all allowed constellations. As a second choice, the user can state the constraints by using a constraint language having the same syntax as the standard "language" in our spreadsheet environment. For example, a conditional statement of type $if < cond > then < val1 > else < val2 >$ is expressed in Microsoft Excel as `IF(cond;val1;val2)`. Currently our constraint language supports standard arithmetic, relational, and logic operators $(=, +, -, *, /, >, <, NOT, OR, IF...)$ for integer variables. In addition, some constraints that can typically be found in Constraint Satisfaction problems like *AllDifferent* - meaning that all variables of a given set of variables must take a different value - where introduced although there is no direct correspondence to a similar standard spreadsheet function. Note that the developer of the spreadsheet application has to be aware of the fact that formulated restrictions are *declarative* constraints rather than *functions*, which he can use in ordinary spreadsheet applications.

The developer states the constraints as plain text which is then parsed online, immediately translated into the internal representation of the underlying constraint solver, and dynamically added to the problem space. The individual constraints can be annotated with a name and an explanatory text. The constraint name and the parse errors are prompted in cases where the constraint parser detects syntactic or semantic errors, e.g., a reference to an undefined variable name or a type error. The explanatory text will be used if - during the propagation process of the user inputs - the solver detects a constraint violation. In these cases, the user will be prompted the information which of his/her selections caused a constraint violation and which value selections have to be

undone or changed. Note however that the frequency of such situations is small because the underlying constraint propagator is capable of inferring the set of still consistent (allowed) values for the remaining variables after each user input.

All the information of the constraint problem is simply entered at arbitrary places in the spreadsheet application[2]. In the *Management* area (5) we define which areas contain the actual constraint satisfaction problem which allows us to arbitrarily design the layout of the application.

The standard spreadsheet environment is extended with a menu (6) that allows the developer to initiate the search or optimization process. If the constraint solver finds a consistent assignment the result is displayed in the output area of (1). In the example from Figure 1, the solver can derive fitting values for the operating system, the basic model and so forth. These computed values can then be referenced from within other parts of the spreadsheet application. During the interaction process where the user sequentially enters the input values, the solver can immediately propagate the effects of the new input and remove inconsistent values from the domains of the remaining variables. At any stage, the user can initiate the search process such that the constraint engine computes values for the remaining problem variables where no user input was given. As a net result for the developer, the trial and error interactive development style of spreadsheet applications without (lengthy) compilation, test, and debug runs is preserved. On the other hand - when *using* the application - high interactivity with immediate feedback is possible such that the user of the system can explore different solution variants and scenarios.

## 3 The CsSolver system

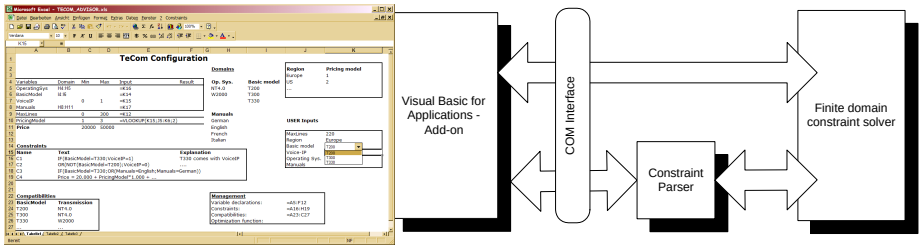


**Fig. 2.** Architecture overview.

**Architecture of the CsSolver system:** In Figure 2, we sketch the different components that make up the CsSolver system. The core of the system is formed by a small object-oriented finite-domain constraint solving library that is built in the tradition of the first approach from [9] that combines Constraint Programming and Object-Orientation using C++, i.e., variables, constraints as well as search goals are classes of the framework. The main advantages of such an approach that can also be found in commercial constraint solvers like ILOG

[2] Typically, the constraint problem is only a part of a larger application that e.g., includes reports or diagrams.

Solver[3] are twofold: First, constraint solving functionality can be easily incorporated in an (object-oriented) application without needing a bridge to some e.g., Logic Programming environment. Second, the inherent extensibility of object-oriented systems can be exploited: The components of the framework can be subclassed for the given application domain, e.g., by introducing a specific type of constraint of new search heuristics. The library implements a forward-checking and backtracking search algorithm and a variant of the *AC-3* [14] algorithm for maintaining arc-consistency and reducing the search space.

The library was developed in *C#*, a programming language from Microsoft's *.Net Framework*[4]. *C#* is an object-oriented language that is syntactically and semantically similar to *Java* and introduces only a few novel language concepts. The most interesting feature from our perspective is interoperability between applications written in different programming languages which is made possible by compilation of the source code to an intermediate language. Consequently the developed constraint library can be easily included in other applications, which is a key requirement for broad acceptance of knowledge-based systems in industrial environments. Within the CsSolver system, communication between the spreadsheet environment and the core solver is based on a *COM* (Component Object Model) interface. While the standard behavior of the solver that can be accessed via the spreadsheet interface is limited in order to preserve simplicity, the solver's API allows programmers to access the full functionality of the underlying solver engine; thus, specific search heuristics or complex functions can be incorporated by the library developers.

The other components of the CsSolver system are a small add-on program written in the scripting language of the spreadsheet environment that catches the events triggered by the user (e.g., definition of variables or initiation of the search process) and appropriately forwards data to the back-end constraint solver. When the user defines his/her constraint problem using the spreadsheet interface, the application-dependent constraints have to be transformed to the internal representation of the solver which is accomplished by a compiler that parses the expressions and issues calls to the solver library accordingly.

**Constraint Satisfaction and Object Orientation:** In [9], a C++ library (ILOG Solver) was presented that for the first time combined object-oriented programming with concepts from Constraint Logic Programming (CLP) like variables, constraints and backtracking search. Later on also other systems like CHIP[5], that were originally based on Prolog, were available as libraries for imperative languages (C and C++). For the CsSolver system we adopted a similar approach using the *C#* language, whereby at the moment only a subset of the functionality of commercial software packages is available, e.g., no floating-point arithmetic.

A fragment of how to use the constraint library is sketched below:

```
1: CsSolver s = new CsSolver();
2: CsVar a = s.createIntVar(0,3,"a");
3: CsVar b = s.createIntVar(0,3,"b");
4: s.addConstraint(a != b);
5: s.addGoal(new CsInstantiate(s,a));
```

[3] http://www.ilog.com
[4] http://www.microsoft.com/net
[5] see http://www.cosytec.fr

```
6: s.addGoal(new CsInstantiate(s,b));
7: if (s.solve())
8:  Console.WriteLine("Found Solution");
```

After initialization of a new solver object that manages the constraint networks (1) two constrained variables with a finite domain [0..3] are declared. In line (4) a constraint is declared that in a consistent solution these two variables must have a different value. Note that we can utilize the *operator overloading* feature of the *C#* language such that an intuitive representation of the constraints is possible. In the following lines, the search goals (instantiation of variables) are declared and finally the solution search is initiated.

The important benefits of such an implementation are as follows (see also [9]): Since all the CSP related concepts are implemented as classes, the framework is intrinsically extensible for specific application domains that require e.g., special sorts of constraints, heuristics, or actions to be taken during the search process. For these cases, the particular behaviour for the application domain can be implemented by subclassing the framework's classes. On the other hand, the classes of the framework can certainly be members of other application-specific classes such that the constraint variables are instantiated for each application specific object.

### 3.1 Evaluation

The business scenarios addressed with the CsSolver system comprise e.g., on-site sales configuration and quotation, i.e., the users of the spreadsheet application are sales representatives that - possibly together with a customer - interactively tailor the solution to the customer's needs. Our experiences show that for these cases, companies tend to equip their sales representatives with sales force automation tools that are developed for the given application domain. These tools are typically implemented by incorporating an intelligent reasoning system like a rule engine, a Logic Programming environment, or a constraint reasoner. Within rule-based systems, the knowledge is encoded in terms of *if-then* rules which are on the one hand easy to understand for non-programmers (in cases where there is an intuitive notation) but on the other hand have shown to cause severe maintenance problems [1]. Moreover, these systems are limited in their expressiveness, optimization is only based on heuristics, and they are limited with regard to explanation both in cases where no solution can be found as well as in cases where the customer wants to know the reasons why the system came up with a particular solution. Logic Programming systems like *Eclipse*[6] offer a complete programming environment including constraint solving capabilities; constraint-based optimization tools like ILOG Solver provide a whole suite of different algorithms for complex, large-scale optimization problems. In principle, the latter two classes of systems can theoretically be used as a back-end for our CsSolver environment. However, when using Logic Programming environments, specialized programming staff and skilled knowledge engineers are required; in addition, the user interface is typically developed using a traditional programming language which can lead to integration problems if there are several development teams involved. On the other hand, commercial optimization tools are in many cases heavy-weight libraries that exceed the functionality that

[6] http://www.icparc.ic.ac.uk/eclipse.

is required in our targeted application domains. Finally, given the small fraction of utilized functionality, the typically high license costs for these advanced systems may also hamper the wide-spread application of these systems.

In many cases we encountered limited acceptance for these sales-force automation tools by the sales personnel because of proprietary user interfaces: The terminology used in the interface is related to the underlying reasoning technology and therefore not understandable for the domain expert. In addition the problem solving process in many cases requires complex interactions where the user has to jump back and forward through different input screens. In fact, we encountered cases where the sales personnel did not use the provided sales-force automation tools but rather built small stand-alone systems based on spreadsheets by themselves.

Another method of solving optimization problems which is supported by some spreadsheet applications are *Operations Research* techniques like linear optimization. In fact there are many problems that can be more efficiently modeled and solved using different variants of classical mathematical algorithms like the Simplex method. We see our work as complementary as these methods require a certain amount of mathematical background knowledge from the users that model the problem; moreover, the expressiveness is in many cases limited to equations and inequalities. The Constraint Satisfaction approach in contrast has its advantages as it relies on a very simple basic model and the relations among the variables can be intuitively modeled using a constraint language. While mathematical methods in many cases are able to compute *one* optimal solution, the search method used in our approach gives the user the possibility to compute multiple alternative solutions for cases where the optimization criterion cannot be expressed as purely mathematical function as the (more or less vague) user preferences have to be taken into account. Finally, the AI-based CSP approach enables the system to generate explanations, why a certain solution was found or no solution was found, which is an important factor in interactive applications.

## 4 Related work

In [10], Renschler describes a practical application for the configuration of radio base stations using a spreadsheet-like interface. From the end user's perspective, this approach is similar to ours regarding the application domain of product configuration: the end user can interactively select the desired features and gets immediate feedback on incompatible choices whereby this computation is based on the integration of an underlying *Constraint Logic Programming* environment and a finite domain constraint solver. There are, however, several important differences when we compare their work with ours: First, although the interaction style is called "Configuration spreadsheet", in our opinion, the actual implementation resembles the main characteristics of spreadsheet only in the sense that the user interface elements are arranged in tabular form, no hard-coded sequence of interaction is required, and immediate feedback is given. The user of the system can not, however, insert any formulae or use additional input or output fields. Moreover, the user interface is extended with domain and search-specific input controls which are unknown in standard spreadsheet applications and therefore require a certain amount of knowledge on the reasoning mechanism by the system's user. Their approach addresses the aforementioned paradigm mismatch of easy-to-use spreadsheets and constraint programming only from the end user's

perspective (using the spreadsheet as mere frontend) but does not solve the problem of program development for the knowledge base.

The *Knowledgesheet* approach presented in [5] is the most similar work compared with our CsSolver system: it relies on an extension of the function-based spreadsheet paradigm with predicates or constraints that are evaluated by an underlying Constraint Programming system. Likewise, with their system, a certain class of constraint problems can be modeled and solved requiring only a limited knowledge of declarative constraint programming concepts like variables or constraints. The main difference compared to our approach is the integration aspect into state-of-the-art spreadsheet environments: while their approach was validated by building a Java-based prototype that interacts with a fully-fledged Constraint Logic Programming (CLP) environment (Eclipse), the CsSolver system is completely integrated into the widespread Microsoft Excel spreadsheet environment and only requires a light-weight object-oriented constraint solver. Moreover, communication between the front-end and the solver in *Knowledgesheet* requires a specific file-based data exchange protocol whereas the CsSolver environment provides a modular API for communication and a flexible way of extending the functionality of the constraint solver. Finally, the constraint language in *Knowledgesheet* follows the notation of the underlying CLP system, whereas we wanted to have a notation for constraints that is as similar as possible to the ordinary spreadsheet functions in the spreadsheet environment. In our opinion, this basically syntactical difference is an important one because in typical applications, the search/optimization problem is always part of a larger (spreadsheet) problem where the standard spreadsheet functionality will be utilized (which is not supported in *Knowledgesheet*).

An even earlier approach to combine the power of interactive software development with spreadsheets and Logic Programming is described in [12]: Based on the idea of Programming-by-Example, the authors propose a spreadsheet-like programming environment (PERPLEX - System) for end users of logic programs. The tabular representation is used to overcome the usability limitations of classical line-oriented interfaces to logic programs: Using a spreadsheet interface, the end-user of the logic program can both incrementally extend the program by posting additional predicates as well as query the contents which are then displayed in tabular form. Compared with our approach, their work only uses the spreadsheet as extended interface for a given logic program, the program itself (i.e., the rules and facts) is still encoded in some logic programming language, like Prolog. Another limitating factor of the system is that the predicates and queries that can be interactively posted have to be expressed in the Logic Programming language of the underlying system.

Despite their wide-spread use in various application domains, e.g., Management Information or Decision Support, many facets of spreadsheet programs and spreadsheet development are still unexplored. The main research efforts in that direction aim at applying established software engineering practices to the spreadsheet development, see e.g., [6] or [11]. The significant demand for adequate development support is driven by the fact that most spreadsheet programs are developed by people who are not IT-professionals, which results is error-prone systems and severe maintenance problems. Complementary, current research efforts exist, that try to apply the intuitive spreadsheet interaction paradigm for general software development based on Visual Programming, see, e.g., [2].

## 5 Conclusions and future work

Our first experiences with the developed CsSolver framework showed the applicability of the approach for a wider range of search and optimization problems that can be modeled as Constraint Satisfaction Problems like product configuration or time-tabling. The first feedback received from potential end-users of the spreadsheet interface for modeling and solving such problems shows that only a small amount of additional background on Constraint Satisfaction concepts have to be learned in order to be able to design and use the system.

Our future work will be focused on improving the understandability of the reasoning process for the end-user. This includes both *explanation* of a solution or explanation of a failure [7] and repair support [3].

## References

1. V.E. Barker, D.E. O'Connor, J.D. Bachant, and E. Soloway. Expert systems for configuration at Digital: XCON and beyond. *Communications of the ACM*, 32, 3:298–318, 1989.
2. M. Burnett, J. Atwood, R. W. Djang, J. Reichwein, H. Gottfried, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.
3. A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configurator knowledge bases. In $14^{th}$ *European Conference on Artificial Intelligence (ECAI2000)*, Berlin, Germany, 2000. IOS Press.
4. G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. In B. Faltings and E. Freuder, editors, *IEEE Intelligent Systems, Special Issue on Configuration*, volume 13,4, pages 59–68. IEEE, 1998.
5. G. Gupta and S.H. Akhter. Knowledgesheet: A graphical spreadsheet interface for interactively developing a class of constraint programs. In *Practical Aspects of Declarative Languages, Lecture Notes in Computer Science 1753*, pages 308–323. Springer Verlag, 2000.
6. T. Isakowitz, S. Schocken, and H. C. Jr. Lucas. Toward a logical/physical theory of spreadsheet modeling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.
7. U. Junker. Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, 2001.
8. B.J. Pine, S. Davis, and B.J. Pine II. *Mass Customization : The New Frontier in Business Competition*. Harvard Business School Press, 1999.
9. J. F. Puget. A C++ implementation of *CLP*. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.
10. A. Renschler. Configuration spreadsheet for interactive constraint problem solving. In *Practical Applications of Constraint Technology, PACT98*, 1998.
11. B. Ronen, M. A. Palley, and Henry C. Luca. Spreadsheet analysis and design. *Communications of the ACM*, 32(1):84–93, 1989.
12. M. Spenke and C. Beilken. A spreadsheet interface for logic programming. In *Proc. Computer-Human Interaction, CHI98*, 1998.
13. M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2), June, 1997.
14. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.