A Framework to Simplify Usability Analysis of Constraint Solvers

Broderick Crawford[1](✉), Ricardo Soto[1], and Franklin Johnson[2](✉)

[1] Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile
{broderick.crawford,ricardo.soto}@pucv.cl
[2] Universidad de Playa Ancha, Valparaíso, Chile
franklin.johnson@upla.cl

Abstract. Currently, given the complexity of industrial problems, a powerful software is required to solve Constraint Satisfaction Problems. The constraint solvers are a kind of software that are based on a constraint approach. During the last years many constraint solvers have been created, some of them are intricate software and others are libraries to extend the features of a programming language. There are few efforts to have a framework that allows to compare a constraint system and less to allow the usability analysis of the solvers. In most cases, the users of these systems are more concerned about the number of enumeration and propagation strategies that can be used instead of the ease of use of constraint solvers. This paper presents a framework to compare and obtain a simple and objective analysis of the usability of these kind of systems. The paper shows that it is possible to establish comparison in terms of usability, allowing an analysis beyond the simple comparison of their internal strategies.

Keywords: Constraint programming · Constraint solvers · Usability

1 Introduction

The new industrial problems are increasingly difficult to solve, these problems use more complex models with more variables and data. Given the difficulty of these problems is not feasible to solve manually and it is necessary to use complex software to solve them. There is thus a strong need for powerful software tools using a simple user interface. This kind of complex problems are classified as combinatorial problems.

Constraint Programming (CP) [27] is a powerful programming paradigm devoted to the efficient resolution of combinatorial problems. Under this paradigm, a problem needs to be modelled as a Constraint Satisfaction Problem (CSP), which corresponds to a formal representation of the problem. The CSP mainly consist in a set of variables holding a domain and a set of constraints. CSPs are usually resolved by a constraint solver which has a powerful search engine. The search engine finds a proper solution by building and exploring a search tree. The constraint solver has two main process, the enumeration process and the propagation process. The enumeration process is responsible for assigning permitted values to the variables in order to generate partial solutions to be verified, while propagation aims at deleting from domains values that do not lead to any solution. The constraint solvers have different enumeration and propagation strategies, which are used in the resolution process of the problems [29].

During the last years many constraint solvers have been created [32], some of them are intricate software and others are libraries to extend the features of a programming language. Since different kinds of constraint solver are available, in some cases, it is difficult to objectively decide which constraint solver to use. A proper selection of a solver can be vital to a project. The developer must have a constraint solver which suits your needs. In some cases, these can be simple, using a constraint solver as a black box, in which only it is sufficient to enter and tune different parameters. But in other cases the developer will need a flexible system that allows him to develop more complex models, which is not available only by setting the solver.

Usability is defined as a quality attribute to measure the ease with which a user interacts with the system. The system users generally have different levels of expertise and experience. In software engineering, usability is the degree to which a software can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use [1]. A usability analysis may be conducted by a usability analyst. The usability includes methods of measuring usability, such as needs analysis and the study of the principles behind the perceived efficiency of an object. Usability differs from user satisfaction and user experience because usability does not directly consider usefulness or utility [22].

In the literature there are few works based on the usability of the Constraint Programming systems. In most cases, the studies are based on the performance and the number and kinds of strategies that the solvers implement [30] instead of the ease of use of the constraint solvers.

The main idea of this paper is to present a framework to compare and obtain a simple and objective analysis of the usability of the constraint solver. The proposed framework is based on the usability attributes proposed by Nielsen [22]. With this framework we try to measure attributes such as efficiency, ease-of-use, satisfaction, learnability, memorability. They are identified and related to usability scenarios. To test the simple usability analysis framework to constraint solver, we define the specific features needed, and then a heuristic evaluation can be performed using the proposed framework.

This work shows that it is possible to establish an adequate framework for comparing constraint Solvers in terms of usability, allowing an analysis beyond the simple comparison of their internal strategies.

This paper is organized as follows. Section 2 presents the constraint solvers, Sect. 3 presents some principles of usability the framework. Section 4 presents the framework to Simplify Usability Analysis of constraint solvers. The conclusions are outlined in Sect. 5.

2 The Constraint Solvers

A constraint solver is a Constraint Programming System that implements constraint programming to solve CSP [16]. These solvers can integrate a constraint logic language, a constraint programming libraries, and some languages that support constraint programming. The solvers have different enumeration and propagation strategies, which are used in the process of resolution of the problems.

A constraint solver implements an algorithm for solving allowed constraints in conformity with the constraint theory. The constraint solver collects the constraints that arrive. It puts them into the data structure for constraints (constraint store) and then it tests their satisfiability, simplifies and if possible solves them. When used from within a constraint programming language, a constraint solver should be able to perform the following reasoning services: Satisfiability test where evaluates whether it is feasible to satisfy a constraint. Simplification where tries to transform a given constraint into a logically equivalent, but simpler constraint. Determination where evaluates that a variable in a constraint can only take a unique value, Variable projection elimination where eliminates a variable by projecting a constraint onto all other variables [8].

2.1 Classification of CP Solvers

Traditionally, to modelling and solving CSPs logical languages have been used. The logical languages are declarative and efficient. Moreover, there are various efforts to solve CSP using other languages, for which they have implemented specialized library for the management of CP. These efforts to generate constraint programming systems commonly referred as Constraint solvers, have resulted in specialized compilers or libraries to implement Constraint Programming.

The classification of Constraint solvers can be performed by multiple criteria. In this case we classify according to: logic programming languages, libraries to other imperatives languages, or constructed as specific solvers [6].

Constraint Logic programming languages. A brief description of different logic programming language is presented. These languages are classified into two groups; The Glass-Box and Black-Box. So, first the features for each classification will be explained. The distinction between a Black Box and Glass Box is difficult to establish. The Glass-Box [13] languages provide very simple and primitive constraints, whose propagation scheme can be formally specified. The constraints are used to build high-level constraints, for each application. Moreover, the Black-Box languages provide a wide range of high-level constraints whose implementation is hidden from the user. These constraints perform specific tasks very efficiently. In these languages, it is difficult for a user to add new constraints, as these must be defined at a low level requiring a detailed knowledge of the implementation.

Glass-Box Languages: We can define two types of glass-box languages. These differ in the way that constraint propagation may be defined: either using a single form of relational construct called an indexical or by means of special Constraint Handling Rules (CHR) [7].

An indexical is a reactive functional rule of the form X in R where X is a domain variable. R is a set-valued range expression of the form $t_1 \dots t_2$ in which terms t_1 and t_2 denote singleton ranges, parameters, integers, combinations of terms using arithmetical operators or indexical ranges.

This constraint can be seen as an abstract machine for propagation-based constraint solving. It is possible to directly encode most of the higher level FD (finite domain) constraints with this one basic constraint. Traditionally among these languages we can find clp(FD) [15], and SICtus [3].

On the other hand the Constraint Handling Rules is a declarative programming language extension introduced in 1991 [9] by Thom Frühwirth. Originally designed for developing (prototypes of) constraint programming systems, CHR is increasingly used as a high-level general-purpose programming language. A CHR languages can define simplification and propagation over user defined constraints.

The application of consecutively CHRs allows to solve the constraints defined by the user. Originally CHRs were created to simplify the constraint languages, but it has spread to build CP solvers for particular applications and domains.

Black-Box Languages: A Black Box is a system such that the user sees only its input and output data: its internal structure or mechanism is invisible to him [11]. This approach partially addresses the requirement for simplicity since the user does not have to be aware of (or modify or extend) embedded techniques and algorithms. However, a Black-Box constraint solver must have a default configuration that yields in most cases the best behaviour that could be obtained by fine tuning of available options. This can be achieved by making the solver robust. One of the most popular black-box languages are Eclipse [21], Oz [28], Ilog SOLVER [14], B-prolog [33].

Constraint programming libraries. A constraint programming library is a tool kit for developing constraint-based systems and applications. These libraries provide a constraint solver with all characteristics of an imperative language.

The constraint programming library differs from constraint logic programming systems like CHIP [5], Eclipse [2] or SICStus Prolog [3] in some topics such us imperative versus rule-based programming, stateful typed variables and objects versus logic variables and terms, no pre-defined search versus built-in depth-first search.

Constraint programming is often realized in imperative programming via a separate library. Some popular libraries for constraint programming are: Choco [4], Gecode [10], IBM ILOG CP [14], JaCop [17], OscaR [26] among others.

Specific solver systems. These correspond to black-box systems, these systems can be implemented using constraint logic programming languages or Constraint programming libraries. This specialized solvers are closed systems that aim to release the user from the complexity of the problem resolution, and only provide an interface to parameterize them. Some of these Constraint solvers can be Abscon [20], Mistral [12], CPHydra [25].

3 Principles of Usability

Usability refers to the user's experience when interacting with a system. A system with good usability is one that shows all the content in a clear and simple way to understand by the user, this is a fundamental aspect of the software. Jakob Nielsen [22], initially defined five basic attributes of usability:

1. Ease of learning: rapidity with which a user learns to use a system with which has not previously had contact (which user does in a simple, fast and intuitive way).
2. Efficiency: the user can achieve a high level of productivity by knowing how to use a system.
3. Retention in time: that the user easily remember how the system was used if he stops using it for a while.
4. User error rates: refers to the amount and severity of errors committed by the user. When committing a fault, the system must inform the user and help him solve it.
5. Subjective satisfaction: refers to whether users feel comfortable and satisfied using the system, that is, whether they like it or not (subjective impression).

Nielsen also defines ten principles of usability, which are useful and easy to verify.

1. System visibility: The system must keep users informed of what is happening, through reasonable periodic feedback.
2. Match between system and the real world: The systems must speak the language of the users, with words, phrases and familiar concepts for the user.
3. User control and freedom: Users often choose options by mistake and clearly need to indicate an exit for those unwanted situations without having to go through extensive dialogues.
4. Consistency and standards: Users do not have to guess that different words, situations or actions mean the same thing.
5. Error prevention: A careful design that prevents problems is better than good error messages.
6. Recognition rather than recall: Make objects, actions and options visible. The user does not have to remember information from one party to another. The instructions for using the system must be visible or easily recoverable.
7. Flexibility and efficiency of use. Design a system that can be used by a wide range of users. Provides instructions when necessary for new users without hindering the path of advanced users.
8. Aesthetic and minimalist design. Do not show information that is not relevant. Each piece of extra information competes with the important one and decreases its relative visibility.

9. Help users recognize, diagnose, and recover from errors. To help users, error messages should be written in simple languages, indicate the problem accurately and show a solution.
10. Help and documentation. The best system is the one that can be used without documentation, but always allows a help or documentation, this information should be easy to find, directed to the tasks of the users, list the concrete steps to do something and be brief.

Usability also used the heuristic evaluation (HE), HE is a usability engineering method "for finding usability problems in a user interface design by having a small set of evaluators examine the interface and judge its compliance with recognized usability principles (the "heuristics")". This method uses evaluators to find usability problems or violations that may have a deleterious effect on the ability of the user interact with the system. Typically, these evaluators are experts in usability principles, the domain of interest, or both. Nielsen and Molich [24] described the HE methodology as "cheap", "intuitive", "requires no advance planning", and finally, "can be used early on in the development process". Often this methodology can be used in conjunction with other usability methodologies to evaluate user interfaces [23].

4 The Proposed Framework

In the literature, there are not studies about usability in constraint solvers, just some works comparing constraint languages and constraint solvers [6,18,31] have been presented. But in all cases, there is an inherent difficulty in trying to compare different systems built it in different environments, languages, and paradigm. For this reason, we propose a framework to simplify usability analysis of constraint solvers and make an objective evaluation based on the usability attributes proposed by Nielsen.

To develop a general framework for different constraint solvers, we must establish some broad criteria, which are not subject to specific conditions. Furthermore comparing different solvers is subjected to factors such as differences in modelling for each solver, different settings, among others [19,32]. Thus we do not consider runtimes, or number of backtracks. we will only establish a simple and clear mechanism to measure constraint solver according to the specific usability features that the evaluator needs.

Our framework is based on the usability attribute proposed by Nielsen and the use of heuristic evaluation to test the usability using a standard test. This framework provides a methodology based on 2 phases; *Design phase*, at this stage the modelling of the test is carried out and the *Evaluation phase*, it is the application of heuristic evaluation. The framework is presented in Fig. 1.

The *Design phase* suggests focusing on starting by modelling the usability of the constraint solver, using the usability measurement model based on a three-level hierarchy. This model defines the usability of constraint solver in terms of: Criteria, metrics and attributes. This can be seen in Fig. 2.

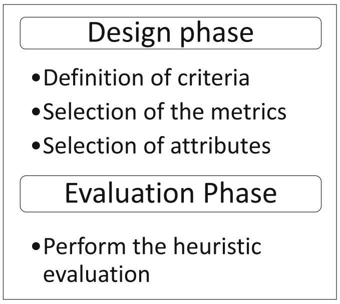

Fig. 1. Phases of the framework for CP solvers

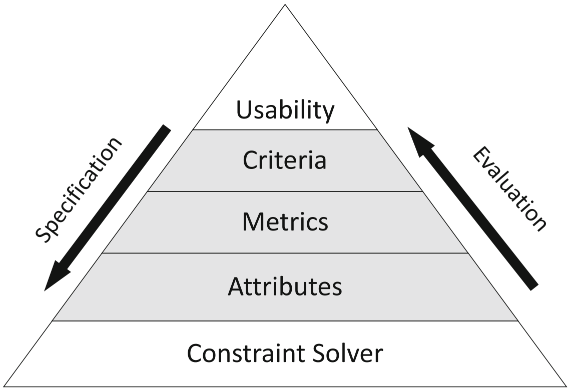

Fig. 2. Model based on a three-level hierarchy

First level: Definition of evaluation criteria. The criteria constitute the parameters for the evaluation of usability at the highest level (first level). The use of criteria refers to the use of a set of specific identifiers and primary characteristics, which allow a critical examination of a Constraint solver.

Second level: Definition of evaluation metrics. In this context, they are defined as two types of arguments; Attribute and measure of the attribute.

Third level: Definition of evaluation attributes. They are metrics that require the definition of attributes and must be declared qualitatively or quantitatively.

The Fig. 2 allows us to visualize the relationship between the levels and processes of usability evaluation. If a solver has more heuristic evaluations made using the three-level hierarchy model, it will have greater usability. And from this new specifications could be generated to modify the usability attributes of the solver. This last part is intended for future corrections and improvements that can be proposed to a solver.

4.1 Design Phase

In this phase, we determine the parameters for the measurement of usability. We have defined a set of criteria that allow us to evaluate the usability, for each criterion a metric is applied and for each metric an attribute is measured. For the specific case of constraint solver, we have taken some criteria defined in the previous section and adapted to be measured according to the features of the solvers. We have defined the following criteria; *Learning* (in Table 1), *Contents* (in Table 2), *Operability* (in Table 3), *Attractiveness* (in Table 4), and *Satisfaction* (in Table 5).

4.2 Evaluation Phase

At the end, of the design phase and once the evaluation guide has been defined, the heuristic evaluation can then be performed. In order to locate problems associated with usability, a heuristic evaluation can then be applied, which allows knowing in depth the constrain solver both functionally and its errors or possible improvements.

Table 1. List of metrics, and attributes associated to criteria Learning

Criteria	Metrics	Attribute
Learning	Ease of learning	Predictive
		Familiarization
		Synthetic
	Help	Consistency between the quality and quantity of help
		Context sensitive help
	Documentation	Access to documentation / tutorials
		Sufficiently explanatory and brief
	Effectiveness	Create a CSP model without help / documentation
		Solve a CSP without help / documentation
		Minimization of execution errors

Table 2. List of metrics, and attributes associated to criteria Contents

Criteria	Metrics	Attribute
Contents	Content to control the enumeration	Data type and data structures
		Variable selection heuristics
		Value selection heuristics
	Content to control the propagation	Definition of constraint
		Create propagators
	Content for cooperation	Integration and portability
		Input/output mechanisms

Table 3. List of metrics, and attributes associated to criteria Operability

Criteria	Metrics	Attribute
Operability	Modelling facility	Definition of constraint
		kind of constraint
		Facility of reification
		Facility to define propagators
		Facility to define value selection heuristics
		Facility to define variable selection heuristics
	Ease of running a model	By command line
		By code embedding
		By call of functions
	Easy to use	Ease of installation
		Simple and clear language
		Allows selection for operating parameters
	Error tolerance	Self-exploratory error messages
		Minimize recovery time
		Facilitates the correction to continue
		Detection and warning of entry errors
	Understanding	Interpretable interface functions
		Clear explanation of input/output actions
		Ease to understand the sequence of answers
		Short messages and simple language
		Clear functions that facilitate recall

Table 4. List of metrics, and attributes associated to criteria attractiveness

Criteria	Metrics	Attribute
Attractiveness	Attractiveness of the interface	Aesthetically pleasing
		Consistent presentation
		Presentation of results in text and graphics
		Combination of color/backgrounds
	Customize	Customization of elements for modelling
		Customizing elements to run CSP
		Changeable elements in the interface

Table 5. List of metrics, and attributes associated to criteria Satisfaction

Criteria	Metrics	Attribute
Satisfaction	Reliability	Texts and messages are easy to read
		Simple and pleasant overall appearance
		Allows access to help comfortably
		The results are clearly presented
	Acceptability	Update mechanisms
		Multiple functionalities
		On-line support

Heuristics evaluation tasks. To heuristic evaluation to be effective, that is, the greatest possible number of usability errors are found, it is recommended to perform a series of tasks, which are presented below:

- Study previously the constraint solver to familiarize yourself with it.
- Determine the usability parameters established in the design phase.
- Define, for each of the parameters, a series of questions to determine if they are met. Make an evaluation guide where each of the questions has the frequency with which the problem appears as well as its impact. The proposed criteria to estimate the impact of each of the questions is shown in the Table 6.
- Perform the heuristic evaluation of the tool using the guide.

Then it is proposed to make a selection of users. The selection of users is a fundamental element in the evaluation process. In the selection can be considered users with knowledge of the constraint solver, given the specific use of these software. Finally, in the framework a review and analysis of data is proposed. A systematic analysis of the data must be done in order to prepare a report detailing the problems and possible solutions applicable to solver.

Table 6. Definition of impact

Impact	Explanation
Low (1)	Although it is recommended that the statement be fulfilled, its non-compliance does not imply confusion or error in the user. It would not give important usability problems.
Medium (2)	Failure to comply can cause not very serious problems of usability although it is convenient to solve them since it would facilitate the operation of the system.
High (3)	Produces significant problems of understanding and functionality in the system so it is essential that the problem is solved. It can cause serious usability problems

5 Conclusion

Currently, there is a variety of constraint solvers. These can be of different types and cover different objectives. For this reason, it is difficult for a specialist to decide which solver to use in a particular project. On the other hand, there are no works that propose to evaluate any of these systems in terms of usability. For this reason, we propose a framework to simplify usability analysis of constraint solvers and make an objective evaluation based on the usability attributes proposed by Nielsen.

We presented a classification of the different constraint solvers. Later we presented the structure of the proposed framework. This framework is characterized by using two phases: a design phase, in which it is modelling the usability of the constraint solver, using the usability measurement model based on a three-level hierarchy. This model defines the usability of constraint solver in terms of: Criteria, metrics and attributes. Later it is defined the evaluation phase which consists in conduct the experimental evaluation, in this phase a heuristic evaluation is performed.

Although no more extensive work has been done, this framework aims to provide a general guide to analyse the usability of constraint solvers, delivering a series of criteria, metrics and attributes specially adapted for this type of system. In future work, the tests should be done and see if they are significant for the use of solvers.

Acknowledgments. Broderick Crawford is supported by Grant CONICYT/FONDECYT/REGULAR/1171243, Ricardo Soto is supported by Grant CONICYT/FONDECYT/REGULAR/1160455

References

1. ISO 9241–11: Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11: Guidance on usability. International (1998)
2. Apt, K.R., Wallace, M.: Constraint Logic Programming Using Eclipse. Cambridge University Press, New York (2007)
3. Carlsson, M., Mildner, P.: Sicstus prolog - the first 25 years. CoRR abs/1011.5640 (2010)
4. choco Team. choco: an open source Java constraint programming library. Research report 10-02-INFO, École des Mines de Nantes (2010)
5. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Herold, A.: The CHIP system: constraint handling in Prolog. In: Lusk, E., Overbeek, R. (eds.) CADE 1988. LNCS, vol. 310, pp. 774–775. Springer, Heidelberg (1988). https://doi.org/10.1007/BFb0012892
6. Fernández, A., Hill, P.: A comparative study of eight constraint programming languages over the boolean and finite domains. Constraints **5**(3), 275–301 (2000)
7. Fruhwirth, T.: Theory and practice of constraint handling rules. J. Logic Program. **37**(1–3), 95–138 (1998)
8. Frühwirth, T., Abdennadher, S.: Principles of constraint systems and constraint solvers (2005)

9. Frühwirth, T., Raiser, F. (eds.) Constraint Handling Rules: Compilation, Execution, and Analysis, March 2011
10. Gecode Team. Gecode: Generic constraint development environment (2006). http://www.gecode.org
11. Gent, I.P., Jefferson, C., Miguel, I.: MINION: a fast scalable constraint solver. In: Proceedings of ECAI 2006, Riva del Garda, pp. 98–102. IOS Press (2006)
12. Hebrard, E., Siala, M.: Mistral 2.0. LAAS-CNRS, Universite de Toulouse, CNRS, Toulouse, France, XCSP3 Competition (2007)
13. Hentenryck, P.V., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). J. Logic Program. **37**(1–3), 139–164 (1998)
14. IBM company: Ibm ilog cp (2006)
15. Jaffar, J., Michaylov, S., Stuckey, P.J., Yap, R.H.C.: The CLP(R) language and system. ACM Trans. Program. Lang. Syst. **14**(3), 339–395 (1992)
16. Mariott, K., Stuckey, P.: Programming with Constraints: An Introduction. MIT Press, London (1998)
17. Kuchcinski, K., Szymanek, R.: Jacop library user's guide (2010). http://jacopguide.osolpro.com/guideJaCoP.html
18. Lazaar, N., Gotlieb, A., Lebbah, Y.: A CP framework for testing CP. Constraints **17**(2), 123–147 (2012)
19. Lecoutre, C., Roussel, O., van Dongen, M.: Promoting robust black-box solvers through competitions. Constraints **15**(3), 317–326 (2010)
20. Lecoutre, C., Tabary, S.: Abscon 109 A generic CSP solver (2006)
21. Niederliński, A.: A gentle guide to constraint logic programming via eclipse (2012)
22. Nielsen, J.: Usability 101: Introduction to usability. Nielsen Norman Group, 4 January 2012
23. Nielsen, J., Molich, R.: Teaching user interface design based on usability engineering. SIGCHI Bull. **21**(1), 45–48 (1989)
24. Nielsen, J., Molich, R.: Heuristic evaluation of user interfaces. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 1990, pp. 249–256. ACM, New York (1990)
25. O'mahony, E., Hebrard, E., Holland, A., Nugent, C.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Iris Conference on Artificial Intelligence and Cognitive Science (2008)
26. OscaR Team. OscaR: Scala in OR (2012). https://bitbucket.org/oscarlib/oscar
27. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
28. Smolka, G.: The development of Oz and Mozart. In: Van Roy, P. (ed.) MOZ 2004. LNCS, vol. 3389, p. 1. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31845-3_1
29. Soto, R., Crawford, B., Olivares, R., Galleguillos, C., Castro, C., Johnson, F., Paredes, F., Norero, E.: Using autonomous search for solving constraint satisfaction problems via new modern approaches. Swarm Evol. Computat. **30**, 64–77 (2016)
30. Soto, R., Crawford, B., Palma, W., Galleguillos, K., Castro, C., Monfroy, E., Johnson, F., Paredes, F.: Boosting autonomous search for CSPs via skylines. Inf. Sci. **308**, 38–48 (2015)
31. Tulácek, M.: Constraint solvers, bachelor thesis, Charles university in Prague (2009)

32. Wallace, M., Schimpf, J., Shen, K., Harvey, W.: On benchmarking constraint logic programming platforms. Response to Fernandez and Hill's "a comparative study of eight constraint programming languages over the boolean and finite domains". Constraints **9**(1), 5–34 (2004)
33. Zhou, N.-F.: The language features and architecture of B-Prolog. Theory Pract. Log. Program. **12**(1–2), 189–218 (2012)