Towards a Formalism of Configuration Properties Propagation

David Fabian[1] and **Radek Mařík**[2] and **Tomáš Oberhuber**[3]

Abstract. Software configuration often studies two issues: firstly, how to merge various software components together to create a program with a fixed structure that fits the requirements, and secondly, how to effectively set up the remaining (usually installation specific) configuration options when deploying the program. Nowadays, the user demands a simple and well arranged way to set up these options, possibly through a graphical user interface (GUI). There are various tools designed to assist the user with these tasks. In this paper, a general multi-platform configuration tool Freeconf is introduced. Our technique to simplify a GUI, which has been incorporated into Freeconf, is described. This technique is based on a set of properties that allow splitting the universe of configuration options into several categories with a clear semantics and rules that control the dynamics of options distribution to these categories in response to the user's actions. The rules are currently only implemented in the source code of Freeconf as a proof-of-concept without any formal proof of soundness or completeness. Results from the domain of Rule-Based Constraint Programming have been applied in the paper to develop a formal description of the rules.

1 Introduction

While working with a software application, the user usually needs to adjust a working environment to her needs. Nowadays, almost every application lets the user to perform some configuration. The average user often does not understand the background of the program and expects a nice graphical user interface (GUI) to assist her. However, there are many applications (especially in the GNU/Linux environment) that do not have any GUI whatsoever and the only way how to configure them is through configuration text files. A serious problem of these files is that their syntax differs greatly, so the user must learn it first from the documentation. It is also necessary for the user to deeply understand the meaning of various configuration options (configuration keys), their dependencies, and their possible values.

1.1 Configuration Tools

There exist tools that address the above mentioned difficulties. Some are focused on a given domain (or even at one application, Linux kernel is popular) like SmartFrog [2] and LCFG [1, 3] which are designed to administer the installation of large scaled networks of UNIX systems, or MenuConfig [16, ch. 7] which is a primary tool for Linux kernel configuration. Then there exist general tools like LinuxConf [10] and Freeconf [9]. Freeconf is a unique tool as it offers a multi-platform and a multi-desktop configuration of applications of any kind.

1.2 Configuration Properties

Many automatic configuration tools suffer from the overwhelming complexity of the user interface they generate which is a severe problem for the user. One of the possibilities of solving this issue consists in breaking the uniform universe of keys to several categories and providing the categories with an exact semantics. Then, only the keys from a category which is the most interesting to the user at a given moment can be displayed. The solution presented here is based on a set of properties of keys, in other words, on a set of labels that are assigned to every key and determine its membership to a category. In Table 1, there is an example of a set of possible properties for keys categorization. The last property *undefined* represents a set of keys that do not have their value set and therefor could cause problems in the output of the configuration. In other words, this property allows us to describe a form of inconsistency with the instant state of the configuration.

property	meaning
mandatory	The key is important to the configured application and must be filled in.
meaningful	The key has sense in the present settings and its existence is not ruled out by any dependency.
undefined	The key finds itself in an invalid state such as that it has no value set or the value is in conflict with dependencies.

Table 1. Properties used for configuration keys categorization.

Having each key as a feature, this approach resembles feature modeling [7] with extra-functional attributes. The semantics for operators and dependencies, however, is different.

The distribution of keys into categories does not have to be static, some keys can change their roles during the configuration process in response to an outer activity (a dependency event, user input). A mechanism is needed to control the development of the categories. For this, Freeconf uses rules to control the propagation of properties. The current set of rules in Freeconf has been constructed by hand and tested only empirically; it has not been proved, whether the rules are sound or complete. Techniques of the rule-based constraint programming can provide a proof; however, one must first give a formal description to the rules.

[1] Dept. of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, (fabiadav@fjfi.cvut.cz).
[2] Dept. of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague.
[3] Dept. of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague.

The rest of this paper is divided as follows. In Section 2, a brief introduction to Rule-Based constraint programming is presented. Section 3 describes the structure of Freeconf and a set of properties used in the tool. Section 4 introduces the way Freeconf handles propagation of properties. Finally, Section 5 presents a formalized set of rules, and Section 6 concludes.

2 Rule-based Constraint Programming

Rule-Based constraint programming is a special case of a more general constraint programming studied in [4, 8, 12, 13]. Constraint programming is an alternative approach to programming in which a model of a problem is declarative, and then it is solved by general or domain-specific methods. The model is formed by a set of constraints (requirements) on variables so that acceptable variable assignments correspond to solutions to the problem [4]. Following [8], let us assume a sequence of variables

$$X = x_1, \ldots, x_n$$

with respective domains

$$D_1, \ldots, D_n,$$

so each x_i takes its value from the set D_i.

Definition A constraint C on X is a pair $\langle C_R, X \rangle$ where C_R is an n-ary relation over the domains D_i, i.e., $C_R \subseteq D_1 \times \ldots \times D_n$, of solutions to the constraint.

Definition A constraint satisfaction problem (CSP) is a triple $\langle \mathcal{C}, X, D \rangle$ where $X = x_1, \ldots, x_n$ is a finite sequence of variables with respective domains $D = D_1, \ldots, D_n$, and $\mathcal{C}$ is a finite set of constraints, each on a sub-sequence of X.

Definition A solution to the CSP $\langle \mathcal{C}, X, D \rangle$ is an element $d \in D_1 \times \ldots \times D_n$ such that for each constraint $C \in \mathcal{C}$ on a sequence of variables Y it holds $d[Y] \in C$ where $d[Y]$ stands for a projection of d to $Y = x_{i(1)}, \ldots, x_{i(l)}$, i.e., $d[Y] = d_{i(1)}, \ldots, d_{i(l)}$. $Sol(\langle \mathcal{C}, X, D \rangle)$ will denote the set of all solutions to the CSP $\langle \mathcal{C}, X, D \rangle$.

There exist general algorithms for solving a CSP [8, 14] and even entire frameworks, such as Skyblue [15], ECLiPSe [5], and Minion [11]..

The core concept of Rule-Based programming is a rule. Rules are condition-action pairs where the condition part is used to determine whether the rule is applicable and the action part defines the action to be taken [6]. A formal definition of a rule taken from [8] follows.

Definition Assume that $\mathcal{A}$ and $\mathcal{B}$ are sets of constraints such that the constraints in $\mathcal{A}$ and $\mathcal{B}$ are on the variables X with domains D. The expression $\mathcal{B} \leftarrow \mathcal{A}$ is a constraint propagation rule. $\mathcal{A}$ is called the condition and $\mathcal{B}$ the body of the rule. Rules act as functions on CSPs. The application of a rule to a CSP with the variables X is given by

$$(\mathcal{B} \leftarrow \mathcal{A})(\langle \mathcal{C}, X, D \rangle) := \begin{cases} \langle \mathcal{C} \cup \mathcal{B}, X, D \rangle & \text{if} \mathcal{A} \subseteq \mathcal{C}, \\ \langle \mathcal{C}, X, D \rangle & \text{otherwise.} \end{cases}$$

The rule $\mathcal{B} \leftarrow \mathcal{A}$ is correct if $Sol(\langle \mathcal{A}, X, D \rangle) \subseteq Sol(\langle \mathcal{B}, X, D \rangle)$.

3 Freeconf

To help the user to overcome difficulties of software configuration, the project Freeconf has been established. The purpose of this project is to unify the existing configuration process and to assist the user by automatically creating the configuration dialog window similar to one in Fig.1. The configuration process must be smooth from the point-of-view of the user: the dialog must fit nicely into the desktop environment, configuration options must be presented in the logical order, and only what is crucial to fill in should be shown. To better understand how Freeconf addresses these tasks, a short description of its inner structure is presented.

Figure 1. Example of Freeconf generated configuration dialog.

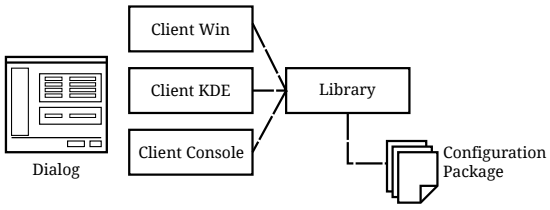

Figure 2. Components of the Freeconf project.

3.1 Structure of Freeconf

The project consists of several components as shown in Fig.2. The most important one is the Freeconf library which contains the entire functionality. The library is shared by graphical client applications (clients), the sole purpose of which is to present a configuration dialog to the user. One can have many different clients, each for one desktop environment. The clients are supposed to be very small (less than a 1000 lines of code), so it should be very simple to create another one. To be able to configure an existing application, the library needs to be provided with an appropriate configuration package. The package is a collection of XML files that semantically describe the configuration file the application understands (native configuration file). It contains a list of all keys, their default values, properties and dependencies, and a rough description of what the resulting dialog should look like. Keys can be organized into configuration sections.

The number of keys can vary depending on the configured application. Usually, configuration packages have tens to hundreds of keys, but some configurations utilize up to thousands of keys like in the case of the Linux kernel.

When a package is loaded, the library constructs three tree structures — a *template tree* for storing key properties, a *configuration tree* for storing values and handling dependencies, and a *GUI tree* for dialog modeling. The trees are interconnected, and one can freely traverse from one node of one tree to the matching node of another tree. Leave nodes correspond to keys and their properties, the non-leave nodes represent configuration sections (there is always a root-section present). When a client needs data for dialog construction, it connects to the library through the client-library interface and obtains various properties for each node. Fig.3 shows the situation. The interface forms a tree which is placed between the GUI tree in the library and the hierarchy of GUI elements (group-boxes, line edits, check-boxes, etc.). The client organizes the GUI elements to another tree that is very closely related to the actual look of the dialog.

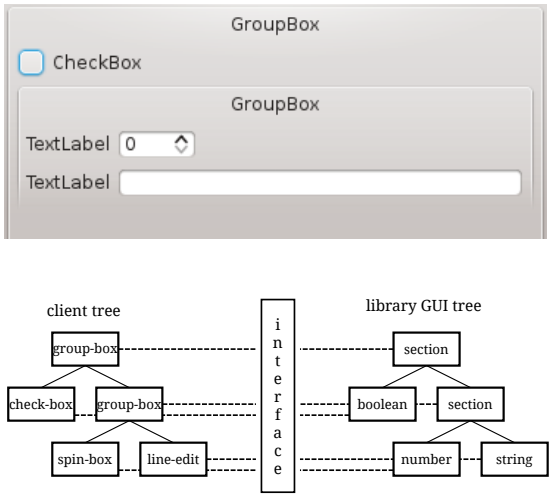

Figure 3. Tree data structures and the client-library interface. The top figure shows how the client tree is transformed into a dialog form.

3.2 Properties in Freeconf

Since the beginning of the project, every client could use the basic set of properties for each key. These basic properties are presented in Table 2.

property	meaning
name	Name of the key.
label	Label for the key.
help	Tooltip help text.
value	Value of the key or default value if no value exists.
type	Type of the key. It can be: boolean, number, string, stringlist, or section.

Table 2. Basic properties of configuration keys.

Every key type adds additional properties, e.g., a *number* can have a *minimum*, a *maximu*m, and a *step* by which the value increments and decrements. String keys usually have a regular expressions associated with them constraining their value.

While the basic set of properties would generally suffice to construct a dialog, the dialog would look overfilled and confusing to the user. That is why another set of properties was added to Freeconf which would enable splitting keys into different categories. Thus, only keys from a specific category can be shown to the user. The current set of properties which extends those presented in Table 1 is summarized in Table 3.

property	meaning
static mandatory	If it is true, the key is mandatory and must be always shown.
static active	If it is false, the key is not visible to the client.
dynamic mandatory	This property can only be set from a dependency handler. If it is true, the key is mandatory and must be shown. This property has no meaning when the static mandatory property is set to false.
dynamic active	This property can only be set from a dependency handler. If it is false, the key does not have sense in the current settings. This property has no meaning when the static active property is set to false.
inconsistent	The key does not have neither a value, nor a default value set and is dynamically mandatory and dynamically active.
empty	The property is only applicable to section nodes. It states whether the section is empty, i.e., all its children are hidden.

Table 3. Additional properties used for keys categorization.

All *static* properties are stored in the package as a part of a template describing the native configuration file, while the *dynamic* ones are a part of a file describing dependencies.

Mandatory property states, whether the key is important or not. Important keys should be visible in the dialog while non-mandatory keys can be hidden, so the dialog becomes less confusing. The *static* version of this property is used for packages with no dependencies or for keys unaffected by any dependency. The *dynamic* version can, as in the current version of Freeconf, override the static state only when the static mandatory property is not false (that means static non-mandatory keys are definite).

Active property has two different semantics. In its *static* version, it is used to prevent the library from announcing the key to the client. In other words, if the property is set to false, the key is virtually commented out. It is easier to disable the key that way than to delete it from the entire package which is non-trivial. The *dynamic* version of this property serves the purpose of ruling out situations that do not have sense (e.g., when the user checks the "no sound" option, setting the "volume" option becomes nonsensical and this option should be left out from the dialog or at least disabled).

Inconsistency is a special situation when the key does not have any value set, but it is important to the configured program. This can happen, especially when the configuration is run for the first time, and there exist keys which do not have default value set by the creator of the package. When this situation occurs, the user has to be told so and must be able to solve the problem with minimum effort. It follows from the above mentioned description of the properties there are situations where inconsistency is acceptable and the user needs not to be alerted (e.g., when the key is statically inactive). In fact, there exists exactly one combination of the properties which needs some user assistance (i.e., the client must be informed) — the key is dynamically mandatory, dynamically active, and inconsistent at the same time. In other situations, such as when the key is only dynamically active but not mandatory, the key is simply left out from the native output (so the native output will always contain only keys

with a defined value).

Emptiness is an important property which naturally arose from the need of hiding non-mandatory keys. The user cannot be distracted by optional keys, so those must be hidden. If there is a section containing only hidden keys, there is no point of displaying it. The empty property can help the client with hiding of unnecessary GUI elements.

4 Properties Propagation

Freeconf maintains properties in connected tree structures as described in Section 3. The Freeconf library must be able to inform the clients about the state of each property in every node the client–library interface announces. For leave nodes, this becomes trivial. For non-leave nodes, however, the state of a property must reflect what is happening in all of node's direct successors.

4.1 Propagation Mechanism

To keep record of the number of properties in children nodes, every section has a set of counters, each bound to a specific property. Counters hold how many times the matching property occurs in the successors. For example, for the inconsistent property the "inconsistent count" counter exists in each section and if it is, for instance, set to two, then there are exactly two children nodes that are inconsistent.

If a counter reaches zero, a message about the change of a property is sent to the client from the affected section. The section must also inform its parent (i.e., another section) about the change, so the appropriate counter of the parent can be adjusted. Similarly, a message must be sent whenever a nullified counter is incremented.

The entire propagation schema can be seen in Fig.4.

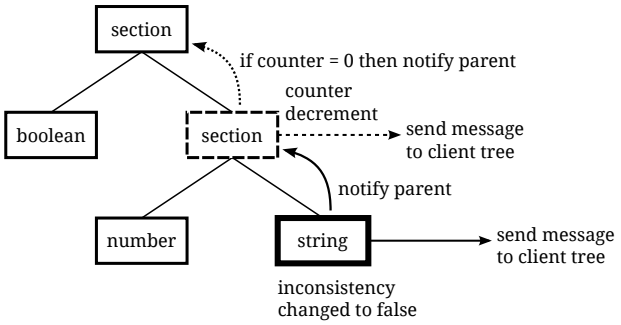

Figure 4. Propagation of properties in Freeconf. Inconsistency of the bold node has been changed. The information is propagated into the parent node (dashed). The property can be further propagated. Every change is sent to the client tree also.

When the state of a property (inconsistency in this case) has been changed, the node notifies its parent about the change. The parent section increments or decrements the matching counter and checks, whether the counter is zero or not. If it is zero, the notification is propagated further up the tree. This leads to the expected behavior in the client since every path leading to an inconsistent key is marked, so the client can render it appropriately. The top-level section (a configuration tab in fact) also knows about the overall state of all keys underneath and it can, for instance, forbid creating the native output until all inconsistencies have been resolved. This method requires a protection mechanism against resending the same message. An obvious solution would be to remember the last state of each property for every node and inform the parent only if the state changes.

For this algorithm to be valid, all counter must be set to the correct value at start time. This is called the initialization phase. All counters are set to zero, and the tree is traversed by depth-first search. Every leave node is evaluated and the existing propagation framework is used to initialize all counter values.

It is also possible to emit a global change, for instance, when the user overrides the mandatory property and enforces showing all keys which are dynamically active. In such a case, all leave nodes are asked to reevaluate their states similarly to the initialization phase. In fact, the initialization phase is a form of a global change.

5 Rules

The above mentioned algorithm was implemented in an ad hoc manner. All property evaluation procedures were tailored to the semantics described in Section 3. The result is a set of rules implemented as condition statements in the source code.

The goal of this section is to bring a formal description of the resulting rules based on definitions from Section 2.

5.1 Formal Description

Let $K = \{k_1, \ldots, k_n\}$ be a set of indices for keys and $S = \{s_1, \ldots, s_l\}$ a set of indices for sections. Let $parent : K \cup S \rightarrow S \cup \{\emptyset\}$ be a mapping returning for each key or section its parent. The symbol of an empty set is returned for the top-level section. All properties of keys will be modeled as Boolean variables. For example, $dynact_x$ will denote a dynamic active property of a key with an index $x \in K$. Together with the properties from Table 3, variables $defvalset_x$ and $valset_x$ will be used to describe the states where a default value and a value have been set to the key, respectively.

Section counters will be modeled as non-negative integer variables. As an example, $inconsistcount_y$ represents the state of an inconsistent counter in a section with an index $y \in S$. If the index is $\emptyset$, no action is performed.

Moreover, there is a Boolean variable called $showallact$ which enables showing even non-mandatory properties (i.e., showing all active keys regardless of the state of the mandatory property). The list of all rules currently used in Freeconf follows.

In the initialization phase, $dynact_x$ and $dynman_x$ are set according to the static version of the properties and $inconsistent_x$ is evaluated for the first time.

$$\begin{aligned} dynact_x &\leftarrow staticact_x \; \forall x \in K \\ dynman_x &\leftarrow staticman_x \; \forall x \in K \\ inconsistent_x &\leftarrow (\neg defvalset_x \wedge \neg valset_x) \wedge \\ &\qquad \wedge dynman_x \wedge dynact_x \; \forall x \in K \end{aligned}$$

When the $valset_x$ variable changes its value, these rules are applied to update inconsistency.

$$\begin{aligned} inconsistent_x &\leftarrow (\neg defvalset_x \wedge \neg valset_x) \wedge \\ &\qquad \wedge \neg inconsistent_x \wedge dynman_x \wedge \\ &\qquad \wedge dynact_x \; \forall x \in K \end{aligned}$$

$$\begin{aligned} \neg inconsistent_x &\leftarrow \neg(\neg defvalset_x \wedge \neg valset_x) \wedge \\ &\qquad \wedge inconsistent_x \; \forall x \in K \end{aligned}$$

Whenever either $dynman_x$ or $dynact_x$ variable changes its value, the inconsistent state of the node must be reevaluated and the parent's counter is adjusted accordingly.

$$inc(inconsistcount_{parent(x)}) \leftarrow$$
$$\leftarrow dynman_x \wedge inconsistent_x \wedge dynact_x \; \forall x \in K$$

$$dec(inconsistcount_{parent(x)}) \leftarrow \quad (1)$$
$$\leftarrow (dynman_x \wedge inconsistent_x \wedge \neg dynact_x) \vee$$
$$\vee (dynact_x \wedge inconsistent_x \wedge$$
$$\wedge (\neg dynman_x \vee \neg inconsistent_x)) \; \forall x \in K$$

If the $inconsistcount_y$ alters, the section must test if it is not necessary to propagate the information further.

$$dec(inconsistcount_{parent(y)}) \wedge \neg inconsistent_y \leftarrow$$
$$\leftarrow (inconsistcount_y = 0) \wedge inconsistent_y \; \forall y \in S$$

$$inc(inconsistcount_{parent(y)}) \wedge inconsistent_y \leftarrow$$
$$\leftarrow \neg(inconsistcount_y = 0) \wedge \neg inconsistent_y \; \forall y \in S$$

The $mandatoryshown_y$ and $activeshown_y$ counters change when a dependency alters $dynman_x$ and $dynact_x$, respectively. It must be also tested whether the static equivalents to the respective properties have not been set to false.

$$\neg dynman_x \leftarrow \neg staticman_x \wedge$$
$$\wedge dynman_x \; \forall x \in K$$
$$\neg dynact_x \leftarrow \neg staticact_x \wedge$$
$$\wedge dynact_x \; \forall x \in K$$
$$inc(mandatoryshown_{parent(x)}) \leftarrow dynman_x \; \forall x \in K$$
$$dec(mandatoryshown_{parent(x)}) \leftarrow \neg dynman_x \; \forall x \in K$$
$$inc(activeshown_{parent(x)}) \leftarrow dynact_x \; \forall x \in K$$
$$dec(activeshown_{parent(x)}) \leftarrow \neg dynact_x \; \forall x \in K$$

The $empty_y$ variable must be reevaluated for each section every time any of its counters (except $inconsistcount_y$) changes.

$$\neg empty_y \wedge dec(sectionshown_{parent(y)}) \leftarrow$$
$$\leftarrow empty_y \wedge \neg(sectionshown_y = 0) \vee$$
$$\vee (showallact \wedge \neg(activeshown_y = 0)) \vee$$
$$\vee (\neg showallact \wedge \neg(activeshown_y = 0) \wedge$$
$$\wedge \neg(mandatoryshown_y = 0)) \; \forall y \in S \quad (2)$$

$$empty_y \wedge inc(sectionshown_{parent(y)}) \leftarrow$$
$$\leftarrow \neg empty_y \wedge ((mandatoryshown_y = 0) \wedge$$
$$\wedge (sectionshown_y = 0)) \vee ((activeshown_y = 0) \wedge$$
$$\wedge (sectionshown_y = 0)) \; \forall y \in S$$

5.2 Weaknesses of Freeconf Design

It can be easily seen that some of the rules are not optimal. For instance, the second rule in 1 could be shortened by leaving out the last occurrence of $inconsistent_x$. In 2, the rules should be mutually exclusive, but it is non-trivial showing the head formulas really behave that way.

Clearly, a problem of the current implementation is the lack of formal description. All condition statements are scattered across the source code, and it is very complicated maintaining them even though the number of the properties is very small. The design is also not very robust since a small change in any of the conditions will render the system non-functioning. This actually happened — one conjunction was overwritten by mistake by a disjunction, and the client started behaving strangely. It was obvious there was a mistake in a condition, but it was difficult to find it.

6 Conclusion

This paper introduces Freeconf, a multi-platform configuration tool, and a technique which reduces the problem of very complex graphical user interfaces that are often generated by automatic configuration tools. The technique is based on splitting configuration options into categories using properties and forming a set of rules that control the dynamics of the evolution of the categories. A set of rules has been proposed to be used in Freeconf to simpify its graphical output. The rules have been implemented in the source code as a proof-of-concept, and it has been empirically verified that the rules work. In this paper, a formal description of the rules has been presented based on the theory of Rule-Based programming. The proof of soundness and completeness of the rules is subject of future work.

7 Acknowledgments

This work was partially supported by the project of the Student Grant Agency of the Czech Technical University in Prague No. SGS11/161/OHK4/3T/14, 2011-13 and Research Direction Project of the Ministry of Education of the Czech Republic No. MSM6840770010.

REFERENCES

[1] Paul Anderson, *LCFG: A Practical Tool for System Configuration*, The USENIX Association, 2008.

[2] Paul Anderson, Patrick Goldsack, and Jim Paterson, 'SmartFrog Meets LCFG: Autonomous Reconfiguration with Central Policy Control', in *Proceedings of the 17th USENIX conference on System administration*, LISA '03, pp. 213–222, Berkeley, CA, USA, (2003). USENIX Association.

[3] Paul Anderson, Alastair Scobie, and Division Of, 'LCFG - the Next Generation', in *UKUUG Winter Conference. UKUUG*, (2002).

[4] K.R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.

[5] K.R. Apt and M. Wallace, *Constraint Logic Programming Using ECLiPSe*, Cambridge University Press, 2007.

[6] Krzysztof R. Apt and Eric Monfroy, 'Constraint Programming viewed as Rule-based Programming', *CoRR*, **cs.AI/0003076**, (2000).

[7] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés, 'Automated reasoning on feature models', in *Proceedings of the 17th international conference on Advanced Information Systems Engineering*, CAiSE'05, pp. 491–503, Berlin, Heidelberg, (2005). Springer-Verlag.

[8] Sebastian Brand, *Rule-Based Constraint Propagation Theory and Applications*, Ph.D. dissertation, 2004.

[9] David Fabian, *System for Simplified Generating of Configurations*, Master thesis, Faculty of Nuclear Sciences and Physical Engineering, Prague, 2011. in Czech.

[10] Jacques Gélinas. Linuxconf homepage, 2005. http://www.solucorp.qc.ca/linuxconf/.

[11] Ian P. Gent, Chris Jefferson, and Ian Miguel, 'Minion: A fast scalable constraint solver', in *Proceedings of ECAI 2006, Riva del Garda*, pp. 98–102. IOS Press, (2006).

[12] Michael Gleicher, 'Practical issues in graphical constraints', in *Principles and Practice of Constraint Programming*, pp. 407–426. MIT Press, (1995).

[13] K. Marriott and P.J. Stuckey, *Programming With Constraints: An Introduction*, Mit Press, 1998.
[14] Nico Roos, Yongping Ran, and H. Jaap van den Herik, 'Combining Local Search and Constraint Propagation to Find a Minimal Change Solution for a Dynamic CSP', in *Proceedings of the 9th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, AIMSA '00, pp. 272–282, London, UK, UK, (2000). Springer-Verlag.
[15] V. Saraswat and P. Van Hentenryck, *Principles and Practice of Constraint Programming: The Newport Papers*, chapter The SkyBlue Constraint Solver and Its Applications, 385–405, Mit Press, 1995.
[16] Sven Vermeulen, 'Linux sea'. http://swift.siphos.be/linux_sea, 2012.