Optimizing Configurations

Tommi Syrjänen[1]

Abstract. In this work we examine the problem of finding optimal configurations. The optimality of a configuration is defined with respect to an optimization function. An optimization function imposes a strict partial order on the set of all configurations of a configuration model and optimal configurations are the minimal elements of this relation. We analyze the computational complexity of the problem and define a class of concise optimization functions for which the optimization problem is in $\Delta_2\mathbf{FP}$. We present a formal rule-based language that can be used to express configuration knowledge and show how we can construct concise optimization functions using the language. As a case study, we formalize a subset of the configuration problem for the Debian GNU/Linux system. We define an optimization function for the system and show how it can be used to implement default host profiles.

1 Introduction

In a software configuration management problem, we are given a set of components each providing a distinct functionality and we want to find a collection of these components, a configuration, that provides the functionality that we want. The set of components and their relationships is called a *configuration model*. In a *configuration task* we are given a configuration model and a set of user requirements and we want to find a configuration that satisfies the requirements. In [11] configurations are divided into three classes:

- *valid* configurations satisfy constraints of the configuration model;
- *suitable* configurations are valid configurations that satisfy the user requirements; and
- *optimal* configurations are suitable configurations that additionally satisfy some optimality criteria.

The configuration management problem is widely researched and [8] presents a recent overview of the field. Most of the effort has been expended on finding valid and suitable configurations and optimization issues have received less attention.

The main aim of this work is to define a formal framework for examining the optimization problem. In most cases there is no single optimal configuration as different users have different needs. Thus, we define the optimality of a configuration always with respect to a specific *optimization function*. The basic idea is to establish a preference relation where we prefer a configuration over another if it gives a better value for the function. This relation is a strict partial order over a finite set so it has at least one minimal element. We say that this minimal element is an optimal configuration. It is possible that there is more than one optimal configuration if the function assigns the same value to many configurations.

A configuration system has to be efficient if it is to be used in practice. We define a class of optimization functions, namely *concise* functions, for which the optimization problem is in $\Delta_2\mathbf{FP}$ if the configuration model is described in a formalism that is in $\mathbf{NP}$. That is, we can solve the problem with a deterministic Turing machine in a polynomial time if we are given an oracle that solves a $\mathbf{NP}$-complete problem in a constant time. The class of concise optimization functions consist of functions that assign each configuration an integral weight less than $a \cdot 2^{n^k}$ where a and k are fixed constants and n is the number of components in the model. This complexity result is proven in Section 7.

Not all configuration formalisms are necessarily $\mathbf{NP}$-complete. Soininen and Niemelä [9] examined the complexity of different constraint types in context of their configuration rule language **CRL** and identified the conditions that allow the configuration problem to remain in $\mathbf{P}$. It seems that most of the interesting cases are $\mathbf{NP}$-complete so we will not give any specific consideration for the problems in $\mathbf{P}$ in this paper.

We use a rule-based language RRL that is a subset of a language RL defined in [10]. Both languages are based on **CRL**. The problem of finding a suitable configuration for this language is proven to be $\mathbf{NP}$-complete in [10].

As a case study, we examine the configuration management of the Debian GNU/Linux system [1]. Debian is a distribution of the GNU/Linux operating system and currently with version 2.1 it has over 2500 distinct software packages that interact with each other in various ways. A package may depend on functionality provided by another one, two packages may conflict with each other, and a package may recommend another package.

In this work we formalize a subset of the Debian configuration problem and show how we can define an optimization weight function for it. In particular, this weight function allows a system administrator to define a set of flexible host profiles that act as default configurations. The complete Debian configuration model is presented in [10].

2 A Configuration Language

In this section we present a declarative rule-based formal language RRL that is a subset of the language RL defined in [10]. The language has a formal semantics that is based on the stable model semantics for normal logic programs [2]. In this work the semantics is presented only informally and the for-

[1] Helsinki University of Technology, Dept. of Computer Science and Eng., Laboratory for Theoretical Computer Science, P.O.Box 5400, FIN-02015 HUT, Finland, Tommi.Syrjanen@hut.fi

mal definition can be found in [10].

The basic language component is an *atom* of the form

$$p(a_1, \ldots, a_n) \ , \tag{1}$$

where p is a predicate symbol and a_1 to a_n are all variables or constants. We will use the convention that all variables start with a capital letter and all constants with a lower case letter. A *literal* is either an atom a or its negation not a. A literal is a *ground literal* if it does not have any variables.

We encode the relationships between atoms using inference *rules* that have two possible forms:

$$h \leftarrow l_1, \ldots, l_n \tag{2}$$

$$\{h_1, \ldots, h_m\} \leftarrow l_1, \ldots, l_n \ , \tag{3}$$

where the atoms h and h_1, ..., h_m form the rule *head* and the literals l_1, ..., l_n form the rule *body*. The rules of the form (2) are called *basic rules* and the rules of the form (3) are called *choice rules*. A basic rule with an empty rule body is called a *fact*. A RRL *program* is a set of rules.

We are interested in finding stable models of RRL programs. Intuitively, a stable model is a set of atoms that satisfies all rules of the program and each atom in it is justified in the sense that it occurs in a rule head that has its body satisfied in the model. A basic rule asserts that if the body of a rule is true, then the head of the rule has to be true also. On the other hand, if the body of a choice rule is true, any number of head atoms $h_1, \ldots, h_m$ can be in the model. Thus, a choice rule gives a justification for a set of atoms to be in a model but it does not force them to be in it. If an atom does not occur as a head of a rule having a satisfied body, it cannot be true in the model.

In addition to rules and atoms a RRL program may also have *optimize statements* of the form

$$\mathsf{maximize}\ \{p(X) : q(X)\} \ . \tag{4}$$

The notation $p(X) : q(X)$ denotes the set of atoms $p(X)$ for which $q(X)$ is also true. The intuitive meaning of an optimize statement is that we want to find a stable model that has as many of the $p(X)$s true as possible. We can also minimize the number of atoms that are in a model by maximizing their complements:

$$\mathsf{maximize}\ \{\text{not } p(X) : q(X)\} \ . \tag{5}$$

If there are more than one maximize statements in a program, they are considered in fixed order with the first one being the least important and the last one being the most important.

Proposition 1 *The existence of a stable model of a ground* RRL *program is* ***NP****-complete.*

Proof. This proposition is proved in [10]. □

This proposition does not necessarily hold for programs with variables as the semantics of a non-ground RRL program is defined in terms of its Herbrand instantiation which may be exponential in size. However, in most practical cases we can use instantiations that are either linear or quadratic.

3 Configuration Models

A *configuration model* CM consists of a set of objects O_{CM} and a set R_{CM} of relationships between objects. The relationships usually include a set of constraints that define what combinations of objects are legal. The most usual constraint types are:

- an *incompatibility* constraint asserts that two components are mutually exclusive;
- a *dependency* constraint asserts that a component needs another component to work properly; and
- a *choice* constraint asserts that we have to choose one or more components of a given set into the configuration.

In this work a configuration model is defined by a RRL program P_{CM} and there is a direct correspondence between legal configurations of CM and stable models of P_{CM}. We can establish this correspondence by defining a bijection *Conf* that maps sets of atoms to configurations and vice versa. In most cases this function is trivial to define. The reason to separate the model from the program that defines it is that the program may have a set of auxiliary atoms that do not have a direct correspondent in the configuration model. It would also be possible to define a configuration model using other formalisms, for example some of the many constraint satisfaction approaches.

A *configuration* c is simply a subset of O_{CM}. A configuration c is *valid* (denoted $c \models CM$) iff there exists a stable model M of P_{CM} such that $Conf(M) = c$. The set of all valid configurations of CM is denoted $\mathcal{C}_{CM}$.

A set U of *user requirements* is represented by a RRL program P_U. A configuration c is *suitable* (denoted $c \models CM \cup U$) iff there exists a stable model M of a program $P_{CM} \cup P_U$ such that $Conf(M) = c$. The set of all suitable configurations is denoted $\mathcal{S}^U_{CM}$.

4 The Debian Configuration Model

The simplified Debian configuration model presented here contains only the most basic features of the Debian configuration system. We use the available software packages as our configuration objects. There are three relationships in the model:

- *dependency*: A package A *depends on* a package B if A cannot be used at all if B is not installed in the system.
- *conflict*: A package A *conflicts with* a package B if A and B are mutually exclusive.
- *recommendation*: A package A *recommends* a package B if B significantly enhances the functionality of A.

The packages are modeled with facts of the form

$$\mathsf{package(P)} \leftarrow \ . \tag{6}$$

The packages have different priority levels that tell how important a package is to the Debian system. We use here only two priority levels, standard and optional. A package is standard if it should be present in all GNU/Linux systems, otherwise it is optional. We take the approach that we denote a standard package P with a fact

$$\mathsf{standard(P)} \leftarrow \ . \tag{7}$$

All other packages are considered to be optional. The relations between packages are modeled with 2-ary predicates depends, conflicts, and recommends.

The configurations are modeled using the predicate in(P) that is true in a stable model exactly when the package P is chosen to be in a configuration. In effect, we define the *Conf* bijection as:

$$Conf(M) = \{\mathsf{P} \mid \mathsf{in}(\mathsf{P}) \in M\} \tag{8}$$

We will take the approach that a package may be added to a configuration only if we can justify it:

$$\{\ \mathsf{in}(\mathsf{P})\ \} \leftarrow \mathsf{package}(\mathsf{P}), \mathsf{justified}(\mathsf{P})\ . \tag{9}$$

We prune out invalid configurations using two constraints:

$$\leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{not\ in}(\mathsf{P}_2) \tag{10}$$

$$\leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2)\ . \tag{11}$$

The user requirements are modeled using two predicates, user-include(P) and user-exclude(P), for packages that the user specifically wants or does not want to have, respectively. The requirements are added as facts in the program as well as the following two constraints:

$$\leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{not\ in}(\mathsf{P}) \tag{12}$$

$$\leftarrow \mathsf{user\text{-}exclude}(\mathsf{P}), \mathsf{in}(\mathsf{P})\ . \tag{13}$$

In the basic model, a package is justified if the user explicitly selected it to be in a configuration, a necessary package depends on it, an included package recommends it, the package is one of the standard packages, or if it belongs to a selected host profile. The host profiles are explained in Section 6.1.

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{user\text{-}include}(\mathsf{P}) \tag{14}$$

$$\mathsf{justified}(\mathsf{P}_2) \leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{15}$$

$$\mathsf{justified}(\mathsf{P}_2) \leftarrow \mathsf{recommends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{16}$$

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{standard}(\mathsf{P}) \tag{17}$$

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{in\text{-}selected\text{-}host\text{-}profile}(\mathsf{P}) \tag{18}$$

Proposition 2 *The existence of a configuration for a Debian configuration model is* ***NP****-complete.*

Proof. By Proposition 1 we only have to prove that the ground instantiation of the configuration model is polynomial with respect to the size of the model.

There are six rules with only one variable in the program and four rules with two variables. The rules with one variable have to be instantiated once for each package in the model and the rules with two variables once for every pair of packages. Thus, if there are n packages in the model, there is a total of

$$6n + 4n(n-1) = 4n^2 + 2n$$

ground instances for the program and its size is polynomial with respect to the size of the model. □

In practice the size of the instantiation is linear since most of the packages are in a relation with less than ten other packages.

5 Optimization Functions

As different users have different requirements, we define the optimality of a configuration always with respect to some optimization function f and there may be many optimization functions defined for a configuration model. We take the approach that optimization functions are defined as weight functions: the function assigns an integral weight for each suitable configuration and we choose the one with highest weight. Formally, we construct a preference relation on the set of all configurations so that we *prefer* a configuration c_1 over c_2 if c_1 has a higher weight than c_2.

Definition 1 *Given a configuration model CM and a weight function $f : \mathcal{C}_{CM} \to \mathbb{N}$, the* preference relation *$P(CM, f)$ is the relation*

$$P(CM, f) = \{\langle c_1, c_2\rangle \mid c_1, c_2 \in \mathcal{C}_{CM}, f(c_1) > f(c_2)\}\ . \tag{19}$$

The Definition 1 imposes a strict partial order on all valid configurations of the model. Given a set of user requirements, we can restrict this relation to suitable configurations.

Definition 2 *Given a configuration model CM, a set U of user requirements, and a weight function $f : \mathcal{S}^U_{CM} \to \mathbb{N}$, the* restricted preference relation *$R(CM, U, f)$ is the relation*

$$R(CM, U, f) = \{\langle c_1, c_2\rangle \mid c_1, c_2 \in \mathcal{S}^U_{CM}, f(c_1) > f(c_2)\}\ . \tag{20}$$

An optimal configuration is a minimal element of the restricted preference relation.

Definition 3 *Given a configuration model CM, a set U of user requirements, and a function $f : \mathcal{S}^U_{CM} \to \mathbb{N}$, a configuration $c \in \mathcal{S}^U_{CM}$ is an* optimal configuration *with respect to f if and only if c is a minimal element of $R(CM, U, f)$.*

Alternatively, we could try to leave out the definition of the restricted preference relation and demand only that a configuration has to be both suitable and a minimal element of $P(CM, f)$. However, this would cause a problem when P has no suitable minimums.

The definitions above do not impose any requirements on the weight function f. In practical applications the weight function should be computationally feasible. We define a class of functions, namely concise weight functions, that has the property that the optimization problem for them is in $\Delta_2\mathbf{FP}$. This will be proved in Section 7.

Definition 4 *A function $f : \mathcal{C}_{CM} \to \mathbb{N}$ is a* concise *weight function if and only if there exist two constants, $a, k \geq 0$, such that for all configurations c, $f(c) \leq a \cdot 2^{n^k}$, where n is the number of components in the configuration model, and f is computable in a polynomial time.*

Both requirements are necessary. The upper bound of the value ensures that we have to consider only polynomial number of potentially optimal weights the time bound ensures that we can compute the value efficiently.

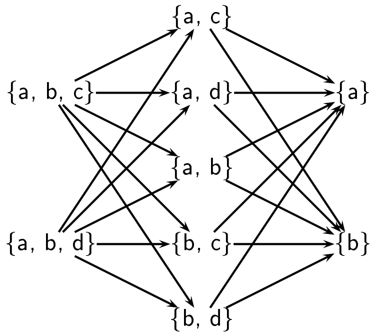

Figure 1. The preference relation of Example 1

Example 1 Let a configuration model CM be defined by the following program:

$$\begin{aligned} \{\ \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\ \} &\leftarrow \\ &\leftarrow \mathsf{c}, \mathsf{d} \\ &\leftarrow \mathsf{not}\ \mathsf{a}, \mathsf{not}\ \mathsf{b} \end{aligned} \tag{21}$$

In addition, let a function $f : \mathcal{C}_{CM} \to \mathbb{N}$ be defined as follows:

$$f(c) = |c|\ . \tag{22}$$

Now CM has nine valid configurations:

$$\begin{aligned} \mathcal{C}_{CM} = \{\ &\{\ \mathsf{a}\ \}, \{\ \mathsf{b}\ \}, \{\ \mathsf{a, c}\ \}, \{\ \mathsf{a, d}\ \}, \{\ \mathsf{b, c}\ \}, \\ &\{\ \mathsf{b, d}\ \}, \{\ \mathsf{a, b}\}, \{\ \mathsf{a, b, c}\ \}, \{\ \mathsf{a, b, d}\ \}\ \} \end{aligned} \tag{23}$$

The preference relation $P(CM, f)$ is presented in Figure 1. There are two configurations in $\mathcal{C}$ that are optimal with respect to f, namely { a, b, c } and { a, b, d }. Suppose that the user wants to have the component d in the configuration so we only allow configurations where d is present:

$$\mathcal{S}^U_{CM} = \{\ \{\ \mathsf{a, d}\ \}, \{\ \mathsf{b, d}\ \}, \{\ \mathsf{a, b, d}\ \}\ \}\ . \tag{24}$$

The unique optimal configuration is now { a, b, d }. $\Diamond$

6 Defining Weight Functions with RRL

We can define concise optimization functions using maximize statements of RRL. The functions are defined in terms of configuration objects by means of *optimization criteria.*

Definition 5 *An* optimization criterion C *is a set*

$$C = \{o_1, \ldots, o_n\} \tag{25}$$

of configuration objects.

We will define the weight function is such way that the more objects $o_1, \ldots, o_n$ are true in a stable model of the RRL program, the higher weight it assigns to the corresponding configuration. If there are more than one criteria, they are ordered in an ascending order of importance and the more important a criterion is, the higher weight it assigns to a configuration. We also allow the criteria to be defined dynamically, during a configuration task.

Definition 6 *Given a configuration c and an ordered tuple $C = (C_1, \ldots, C_k)$ of optimization criteria, the* weight function *$w_k(c)$ is defined as follows:*

$$w_k(c) = \sum_{i=1}^{k} n^{2i} \cdot |C_i \cap c| \tag{26}$$

Theorem 1 *The weight function $w_k(c)$ is concise.*

Proof. Let n be the number of components in the configuration model and k the number of optimization criteria. The weight function $w_k(c)$ is clearly computable in polynomial time. The maximum value of $|C_i \cap c|$ is n so the value of$w_k(c)$ is always strictly less than n^{2k+2} when $n > 1$. Thus, we can assign an upper bound $2^{n^{2k+2}}$. If $n = 1$, then $w_k(c) \leq k$ and we can use $2k$ as a bound so $w_k(c)$ is concise. $\square$

The optimality criteria $C_1, \ldots, C_k$ can be defined with k maximize statements:

$$\begin{gathered} \mathsf{maximize}\ \{\mathsf{in}(X) : C_1(X)\} \\ \vdots \\ \mathsf{maximize}\ \{\mathsf{in}(X) : C_k(X)\} \end{gathered} \tag{27}$$

The main problem of this formulation is that we may use it only when the optimality can be expressed in terms of objects that are in the configuration. In other cases we would like to be able to use a more general weight function. This can be achieved by allowing any literals to be in a criterion and extending the maximize statements to have the form:

$$\mathsf{maximize}\ \{p(X) : q(X) = g(p(X))\} \tag{28}$$

where $g(p(X))$ is an integer-valued function that assigns a weight to each literal $p(X)$. The idea is then to maximize the total weight of the literals. However, the function g has to be defined carefully to ensure that the complete weight function stays concise.

6.1 The Debian Weight Function

The configuration problem of a GNU/Linux system has two important characteristics:

1. the configuration model is large and there are usually hundreds of packages in a configuration; and
2. the GNU/Linux is free software so a configuration with a hundred packages has essentially the same cost than one with a thousand packages.

Because of the first property, configuring the system may be very tedious and it may take many hours if the user has to select packages one at a time. This could be simplified by constructing a tool that allows the user to state in general terms what he or she wants to do with the system. The tool would then automatically generate a configuration satisfying the user's needs. We can do this by defining collections of related packages, *host profiles*, that can be selected in addition to normal packages. The profiles can be defined either by the creators of the distribution or by a local system administrator.

Because of the second property of the problem domain, we would often want to include potentially useful packages into the configuration if there is enough disk space available for them. Installing them at the same time when the other system is configured saves time and trouble in the long run.

We define the optimization criteria for the simplified Debian configuration model using the following principles:

1. maximize the number of packages belonging to a host profile;

2. maximize the number of packages that have standard priority; and
3. maximize the number of packages that are recommend by other packages that are in the configuration.

We can see that these criteria sometimes impose conflicting demands. For example, it is possible that the host profile includes a package that is in conflict with a standard package and satisfying the first criterion makes it impossible to satisfy the second criterion.

We order the criteria so that we prefer adding one package from the host profile over any number of standard packages or recommended packages and one standard package over any number of recommended packages. This is by no means the only possible ordering of the criteria.

Conceptually, during a configuration task we first add packages that are needed to satisfy the user requirements. Next, we add packages that are in the host profile and do not conflict with packages that are already in the configuration. After that, we add all standard packages that do not cause problems. Finally, we try to find a maximal set of packages that are recommended by those already in the configuration and add them to it.

The Debian weight function can be defined with the following maximize statements:

$$\begin{aligned} &\mathsf{maximixe}\ \{\ \mathsf{in(P)}\ :\ \mathsf{is\text{-}recommended(P)}\ \} \\ &\mathsf{maximize}\ \{\ \mathsf{in(P)}\ :\ \mathsf{standard(P)}\ \} \\ &\mathsf{maximize}\ \{\ \mathsf{in(P)}\ :\ \mathsf{in\text{-}selected\text{-}host\text{-}profile(P)}\ \} \end{aligned} \tag{29}$$

The maximize statements are again in an ascending order of importance. The predicate in-selected-host-profile is true when the package in question belongs to an included host profile and the predicate is-recommended is defined as follows:

$$\mathsf{is\text{-}recommended(P_2)} \leftarrow \mathsf{recommends(P_1, P_2), in(P_1)}\ . \tag{30}$$

7 Computational Complexity

In this section we define a set of configuration tasks and analyze their computational complexities. The complexity results suppose that the configuration models are defined with RRL but the results generalize to other **NP**-complete formalisms.

We show that if we have a variable-free configuration model, the validity of a configuration can be checked in a polynomial time with respect to the size of the model. The problem of finding a suitable configuration given a set of user requirements is **FNP**-complete as is the problem of finding a suitable configuration with a specific weight if the weight function f is concise. We show that the problem of finding an optimal configuration is in $\Delta_2\mathbf{FP}$. It is not clear whether the problem is also $\Delta_2\mathbf{FP}$-complete.

The results are stated for variable-free configuration models but they also hold it the size of the instantiation of the model is polynomial, which is true for most practical cases.

Definition 7 *Given a ground configuration model CM and a configuration c, the problem of checking whether $c \vDash CM$ is called* VALID-CONFIGURATION(CM, c).

Theorem 2 *The problem* VALID-CONFIGURATION(CM, c) *is in* $\boldsymbol{P}$.

Proof. The theorem follows directly from Proposition 1. □

Definition 8 *Given a ground configuration model CM and a set U of user requirements, the problem of finding a configuration c such that $c \vDash CM \cup U$ is called* QUERY-CONFIGURATION(CM, U)

Theorem 3 *The problem* QUERY-CONFIGURATION(CM, U) *is* $\boldsymbol{FNP}$*-complete.*

Proof. The theorem follows from Proposition 1. □

Definition 9 *In the problem* QUERY-CONFIGURATION-WITH-WEIGHT(CM, U, f, W) *we are given a ground configuration model CM, a set U of user requirements, a weight function f, and a weight $W \in \mathbb{N}$ and the object is to find a configuration c such that $c \vDash CM \cup U$ and $f(c) \geq W$.*

Theorem 4 *If the weight function f is concise, the problem* QUERY-CONFIGURATION-WITH-WEIGHT(CM, U, f, W) *is* $\boldsymbol{FNP}$*-complete.*

Proof. Follows directly from Theorem 3 and the fact that we can guess the weight of a configuration and verify it in a polynomial time by Definition 4. □

Definition 10 *Given a ground configuration model CM, a set U of user requirements, and a weight function f, the problem of finding a configuration c such that $c \vDash CM \cup U$ and c is optimal with respect to f is called* OPTIMAL-CONFIGURATION(CM, U, f).

Theorem 5 *If the weight function f is concise, the problem* OPTIMAL-CONFIGURATION(CM, U, f) *is in* $\Delta_2\boldsymbol{FP}$.

Proof. By definition [6, p.424], a problem is in $\Delta_2\mathbf{FP}$ when it can be solved in deterministic polynomial time using a **NP**-oracle. As the weight function f is concise, the optimum weight is at most 2^{n^k} for some fixed n and k. We can find the weight of an optimal configuration by using *binary search*. We first check whether there exist a suitable configuration that has a weight 2^{n^k-1} or more. As this is an instance of QUERY-CONFIGURATION-WITH-WEIGHT, it can be done in constant time using the oracle.

Now, depending on the answer we either check whether the optimum weight is over 2^{n^k-2} or over $2^{n^k-1} + 2^{n^k-2}$. Thus, in each step we halve the possible range of the optimal weight and we have to issue $\log_2 2^{n^k} = n^k$ queries so the problem is in $\Delta_2\mathbf{FP}$. □

8 Implementation and Evaluation

The Debian configuration model has been implemented as an extended logic program using the SMODELS system [5] developed in the Laboratory for Theoretical Computer Science in Helsinki University of Technology. The SMODELS system is available at

`http://www.tcs.hut.fi/pub/smodels` .

The full Debian configuration model with preferences was tested using actual data of the main Debian distribution 2.1 with 2255 different packages. The user requirements were generated by choosing randomly a fixed number of packages. The number of user requirements varied between 0–100. In addition, a host profile with 89 packages was created and included in the configuration. The tests were run under Debian GNU/Linux 2.1 on a 450 MHz Intel Pentium III system with 256 MB main memory. The used software versions were `smodels-2.25` and `lparse-0.99.48`.

The results of the tests are shown in Table 1. The times shown are averages over 100 runs. Separate figures are presented for satisfiable and unsatisfiable user requirements. The times show only how long `smodels` took to find an optimal configuration. In addition to this time, `lparse` spent on average 9.1 seconds preprocessing the model. This time is not shown in the figure since in a practical configuration tool this preprocessing has to be done only once for each Debian version. The execution times get faster as the number of user requirements increase because adding new constraints reduces the number of suitable configurations.

Table 1. Evaluation results

#Pck.	Total (s)	Sat. (s)	Unsat. (s)	#Sat
0	2.94	2.94	–	100
5	2.66	2.78	1.43	91
10	2.38	2.69	1.02	82
50	1.80	2.26	1.02	62
75	1.56	2.17	1.02	47
100	1.31	2.11	1.03	26
total	2.11	2.61	1.04	68

9 Related Work

Most of the existing work on optimal configurations has been concentrated in the area of *resource balancing*. In a resource balancing problem we are given a set of *consumers* that each consume a specific resource. There are also several *producers* that generate the resources. The goal is to find a set of producers that generates the resources that consumers use. Each producer has also a distinct *cost* assigned to it and we want to minimize the total cost of producers that are taken into the configuration.

Sabin and Freuder [7] modeled the problem using composite constraint satisfaction. Their basic idea is to achieve optimality by adding a new constraint to the system each time a new solution is found. If the found solution has a cost C, the constraint $cost < C$ is added to the program. This forces the underlying engine to find only better solutions. The approach is complete and it will always find the optimal configuration.

Juengst and Heinrich [3] took another route by proposing a heuristics that would find good, but not necessarily optimal, configurations. They compute the ratio of actual resource demand and highest possible resource supply in a local scope, and try to keep it as low as possible during the configuration process.

Conceptually the approach of Sabin and Freuder fits within our definition of an optimal configuration. The RRL is not well suited for resource balancing programs but if it is extended with weight constraint rules [4], it gains sufficient expressive power to allow a relatively compact representation of the problem.

10 Conclusions and Future Work

We presented a formal definition for optimal configurations. The basic approach is to define an optimization function that imposes a preference relation on the set of all configurations. The optimal configurations are minimal elements of this relation. We defined the class of concise weight functions for which the optimization problem is in $\Delta_2\mathbf{FP}$.

We presented a rule-based language RRL that can be used to define configuration models and showed how optimization functions can be defined with it.

We formalized a subset of the configuration management of the Debian GNU/Linux system and defined a concise weight function for it. The weight function allows a system administrator to define a set of easily customizable host profiles. The basic idea is that first the user requirements are satisfied and then as many default packages are added to the configuration as possible. The full configuration model for the Debian system is presented in [10]. The model has not been incorporated into any existing tool and the next step of this research is to construct a working prototype.

ACKNOWLEDGEMENTS

This work has been supported by the Academy of Finland (project no. 43963) and Helsinki Graduate School in Computer Science and Engineering (HeCSE). We would also like to thank the referees for helpful comments.

REFERENCES

[1] Debian GNU/Linux. Available at: <url:http://www.debian.org>.

[2] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080, Seattle, USA, August 1988. The MIT Press.

[3] Werner E. Juengst and Michael Heinrich. Using resource balancing to configure modular systems. *IEEE Intelligent Systems & their applications*, pages 50 – 58, October 1998.

[4] Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the Fifth Interational Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, December 1999.

[5] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A system for answer set programming. In *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, April 2000.

[6] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley Publishing Company, Inc, 1995.

[7] Daniel Sabin and Eugene C. Freuder. Optimization methods for constraint resource problems. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[8] Daniel Sabin and Rainer Weigel. Product configuration frameworks — a survey. *IEEE Intelligent Systems & their applications*, pages 42 – 49, October 1998.

[9] Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*. Springer-Verlag, January 1999.

[10] T. Syrjänen. A rule-based formal model of software configuration. Research Report A 55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 1999.

[11] B. Wielinga and G. Schreiber. Configuration-design problem solving. *IEEE Expert*, 12 2:49 – 56, March-April 1997.