Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti
Espoo 1999 HUT-TCS-A

A RULE-BASED FORMAL MODEL FOR SOFTWARE CONFIGURATION

Tommi Syrjänen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti
Espoo 1999 HUT-TCS-A

A RULE-BASED FORMAL MODEL FOR SOFTWARE CONFIGURATION

Tommi Syrjänen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-4881-6
ISSN 0783-5396

Picaset Oy
Helsinki 1999

ABSTRACT: In this work we examine the software configuration management problem. We give a short introduction to the current state of the art and present a declarative rule-based formal language for representation of configuration knowledge. As a novel feature, the rule language allows finite existential quantification over the configuration components. We show a translation from the rule language to normal logic programs with stable model semantics. As a case study we examine the configuration management problem for the Debian GNU/Linux system which consists of over 2000 distinct software packages. We examine the current practice to identify the main components of the problem and present a way to formalize them using the rule language. We construct two formal models: one for finding valid configurations and one for diagnosing unsatisfiable user requirements. We show that the decision problem for the configuration model is NP-complete. We present a translator that reads a high-level description of the Debian configuration system and produces the corresponding set of rules. We evaluate the configuration model by using actual data from Debian version 2.1 and a set of randomly generated user requirements. The evaluation shows that the model is computationally feasible.

KEYWORDS: Configuration management, stable model semantics, logic programming, Debian, Linux, diagnostics

Contents

LIST OF SYMBOLS AND NOTATIONS

$\wedge$	logical and
$\vee$	logical or
$\neg$	logical not
$\rightarrow$	logical implication
$\models$	models
$\exists$	existential quantification
$\top$	logical true
$\bot$	logical false
$\in$	containment in a set
M	a model
NP	the set of problems that can be solved in a non-deterministic polynomial time
$O(x)$	at least the order of x
$\mathcal{P}$	a propositional language for expressing conditional relations
P	the set of problems that can be solved in a polynomial time
$\mathcal{V}$	a truth valuation
RL	a rule-based language for configuration management
$h \leftarrow a, \ldots, \mathrm{not}\, b$	a basic inference rule
$\{h\} \leftarrow a, \ldots, \mathrm{not}\, b$	a choice rule
$\exists x(l_1(x) \wedge \cdots \wedge l_n(x))$	an existential literal

ACKNOWLEDGEMENTS

This work contains the results of my work as a research assistant in the Laboratory for Theoretical Computer Science of Helsinki University of Technology. I wish to thank Professors Leo Ojala and Tomi Janhunen and Docent Ilkka Niemelä for giving me the opportunity to work in the laboratory. I am grateful to Docent Niemelä for his comments and advice. I also want to thank my colleagues in the laboratory.

Chapter 1

Introduction

Modern software products are large and complex. They may consist of hundreds or thousands of components that interact with each other in various ways. In addition, each component may have different versions with different requirements. The core of the software configuration management problem is how to ensure that the user receives correct versions of all necessary components of the system.

Software engineering is not the only place where configuration management is used. Today we see more and more configurable products, from furniture to large machinery. As the configuration problems become more complex, we need stronger methods for tackling them.

In this work we examine the configuration management problem for product configuration. The most emphasis is placed on software products but the methods described here may work also in other configuration domains. We will examine some methods that are currently in use and introduce a rule-based language that can be used to express configuration knowledge. As a case study, we examine the configuration management problem of the Debian GNU/Linux system.

1.1 THE DEBIAN GNU/LINUX SYSTEM

Debian GNU/Linux [4] is a distribution of the Linux operating system. The term *Linux* used to denote only the kernel of the operating system but nowadays it is mostly used to mean the whole operating system based on Linux kernel. The book *Linux Application Development* [18, p. 3] defines Linux in following way:

> Linux is a freely distributable, Unix-like operating system that includes a kernel, system tools, applications, and a complete development environment.

Initially users downloaded the kernel and the required applications via the Internet and compiled the programs themselves. The problem with this method was that searching and installing all interesting software was tedious work and the user would need at least some programming skills to get everything running.

Because of the problems, a number of organizations and companies created their own Linux *distributions*. A distribution is a collection of programs that the maker of the distribution thinks are useful to most users. Most distributions are organized as *packages*, each package containing a single program or a document. The user chooses a set of packages from the list of all packages and the installer program of the distribution then installs selected software.

Currently the most well-known Linux distributions are[1]: RedHat [29], Debian [4], Slackware [37], Caldera [2], and SuSe [44].

The Debian distribution was originally created by Ian Murdock in 1993. Initially it was sponsored by Free Software Foundation's GNU project [13]. The current name of Debian GNU/Linux system emphasizes the fact that practically all tools that are used with Linux systems are products of the GNU project.

Currently with version 2.1 there are over 2500 different packages available for Debian and most of them have many different versions. The planned release of version 2.2 in early 2000 will add many more. The system is maintained by over 300 volunteers located all over the globe. The distribution is divided into three parts. The `main` distribution contains all normal packages, the `non-free` distribution contains packages that have license restrictions which prevent them to be included in the main distribution, and the `non-us` distribution contains packages that contain strong cryptography and cannot be exported from United States. The Debian GNU/Linux web site is located at

<`url:http://www.debian.org/`> .

1.1.1 Motivation

The main reasons why this work takes the Debian distribution as a case study and not any of the other distributions are:

1. The relationships between software packages are explicitly described and they are collected in one place. It is possible to generate a configuration model of the system automatically from this information.
2. The scale and diversity of the Debian system makes it well suited for testing new methods of configuration management.
3. The author is most familiar with the Debian distribution.

The base distribution contains currently 2260 different packages and if we add the non-free and non-us distributions we reach a total of 2738 packages. As each package has its own version history, the total number of Debian packages in existence is very large and the configuration management system has to handle all of them correctly. In addition, the packages are written by different authors in different environments and they must somehow be made to work correctly together.

As we see the configuration management in a Debian system is a formidable task and if a formal method can handle it well, it can probably also handle other difficult cases.

[1]A complete list of different Linux distributions can be found at <`url:http://www.linux.org/dist/`>.

1.2 A RULE-BASED APPROACH FOR THE SOFTWARE CONFIGURATION PROBLEM

We will define a rule-based configuration language that can be used to implement configuration models. The relations between objects are encoded by predicates and the constraints are expressed as a set of inference rules. For example, a rule

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{user\_include}(\mathsf{P})$$

asserts that if the user selected a package P, then we have a justification for it. In addition to the inference rules we have a database of facts. For example, if we added a fact

$$\mathsf{user\_include}(\mathsf{a}) \leftarrow$$

to our earlier example we could infer that a package a is justified in the model.

We also need a method to specify optional components. An optional component may be added to a configuration if some condition is fulfilled, but we may also decide to leave it out. For example, a rule

$$\{\mathsf{in}(\mathsf{P})\} \leftarrow \mathsf{justified}(\mathsf{P})$$

signifies that we may add a package P into a configuration if we can justify it but it is no error if we leave it out. In any case, we can only have packages with justifications in the configuration.

As a final feature of the rule language, we will allow finite existential quantifications. There are often situations where many different components provide the same functionality and we want to express a condition that at least one of them is in the configuration. For example, a rule

$$\leftarrow \mathsf{virtual}(\mathsf{V}), \mathsf{in}(\mathsf{V}), \neg\exists \mathsf{P}(\mathsf{provides}(\mathsf{P}, \mathsf{V}) \wedge \mathsf{in}(\mathsf{P}))$$

express the constraint that is violated if there is a virtual package V in the configuration and there is no concrete package P that provides the functionality.

We define the rule language in such way that it can be efficiently translated into logic programs with stable model semantics [12]. This is done because there exists an efficient implementation of the stable model semantics, `smodels`, which we can use to solving configuration problems.

1.3 IMPLEMENTING THE DEBIAN CONFIGURATION MODEL

We can implement the Debian configuration problem with the rule-based language we introduced above.

The basic approach is that we may have any package that we can somehow justify in a configuration.

$$\{\mathsf{in}(\mathsf{P})\} \leftarrow \mathsf{justified}(\mathsf{P})$$

Thus, a configuration is a subset of the set of justified packages. A package is justified if the user requires it or some package that depends on it is selected.

We will restrict the set of valid configurations by imposing a number of constraints. The two most important constraints are dependency and conflict. A package depends on another if the package cannot be used without it and a package conflicts with other package if they are mutually exclusive.

We will model the dependency constraint with a rule of the form

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

The basic approach to the conflict is to use rules of the form:

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

That is, if P_1 conflicts with P_2, both of them may not be in a configuration at the same time.

If the user requirements cannot be satisfied, we would like to have some explanation on what went wrong. In one approach we could try to find the problematic requirement by removing one requirement at the time until a configuration can be found. We will take another approach, however. We add a set of rules and atoms that tell us why the configuration could not be found. For example, we can diagnose the conflict using a rule of the form:

$$\mathsf{in\_conflict}(\mathsf{P}_1, \mathsf{P}_2) \leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2)$$

Traditionally the rule-based configuration formalisms have suffered from maintenance problems: when the number of the rules reaches thousands it may be difficult to see the whole picture. One reason for this is that the rule languages often do not have a declarative semantics and the maintainer has to take all quirks of the inference engine into accord when implementing the model.

As the full Debian configuration model will contain over 50 rules and several thousands of facts, it will not be immune to problems. To help to keep the complexity down we will construct a translator that generates the configuration model automatically from the Debian GNU/Linux configuration files.

1.4 OUTLINE OF THE WORK

In Chapter 2 we present a short introduction to the current state of the art for handling the configuration problem. In Chapter 3 we present a rule-based configuration language which is modeled after one presented in [41] and is based on the stable model semantics.

In Chapter 4 we introduce the Debian Linux system which is used as a case study on the feasibility of our approach to configuration management and in Chapter 5 we formalize it with the rule language.

In Chapter 7 we give more details on the configuration model and in Chapter 8 we evaluate of our configuration model by using actual configuration data from Debian GNU/Linux versions 2.0 and 2.1.

Finally, in Chapter 9 we present our conclusions.

Chapter 2

Configuration Management

In this section we give a brief informal overview of some approaches that have been proposed to solving the configuration management problem. The presentation is mostly based on articles [32] and [3]. The first article gives a broad overview of product configuration frameworks and the second one examines software configuration management.

In a configuration problem we have a complicated product that consists of different components, *configuration objects*, that may interact in complex ways. The collection of objects and relationships between them is called a *configuration model*. A *configuration* is a set of objects of the configuration model. A configuration is *valid* if it satisfies all *constraints* imposed by the model. We may also have a set of *user requirements* for the configuration.

In a *configuration task* we want to find answer to one or more of the following four questions:

- Given a configuration and a configuration model, is the configuration valid for the model. In this work this problem is called VALID-CONFIGURATION.

- Given a configuration model, we want to know whether the configuration model is satisfiable. That is, does the model have any valid configurations. We call this problem EXISTS-CONFIGURATION.

- Given a configuration model and a set of user requirements we want to find one or all configurations that satisfy the requirements. This problem is called QUERY-CONFIGURATION.

- Finally, if we have a configuration model and an unsatisfiable set of user requirements, we would like to know why the requirements could not be satisfied. In this work we call this problem DIAGNOSE-CONFIGURATION.

For example, in a software configuration problem the user requirements might simply be that we want to have an executable program that runs with our hardware and operating system. In a more complex case the software might be composed of a set of modules each providing a single function and we want to find what modules are required to provide all the functionality the user wants.

In a *reconfiguration task* we have an existing configuration and either the configuration model or our user requirements change and we want to find how we can change our configuration so that it satisfies our requirements in the new situation.

A configuration model may have many valid configurations. Often some of these configurations are in some way better than the others. The article [47] divides the set of configurations into three classes:

- a *valid configuration* satisfies all constraints of the configuration model;
- a *suitable configuration* is a valid configuration that also satisfies the user requirements; and
- an *optimal configuration* is a suitable configuration that additionally satisfies some optimality criteria.

During a configuration task we would like to find an optimal configuration. However, we are ready to settle for a suitable one if we cannot find an optimal configuration.

2.1 VERSION MODELS

In general, a configuration problem has two dimensions, the product space and the version space. The configuration objects and their relationships form the *product space* of the configuration model. The *version space* describes the relationships of different versions of the objects. The dimensions are not completely orthogonal since the object relationships may depend on the versions of the objects.

The configuration formalisms can be roughly divided according to two criteria: is it extensional or intensional and is it state or change-based.

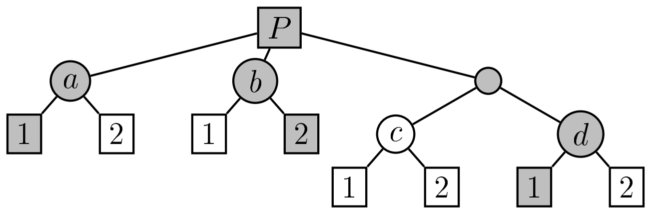
Extensional

A product uses *extensional* configuration management if all configurations and object versions are explicitly constructed and stored in the configuration model. Extensional methods are mostly used by software version control management systems. After a programmer has changed a source code file, the file is checked into the control system where it receives a unique identifier. The user selects a configuration by explicitly selecting the desired versions of each component.

Intensional

A product uses *intensional* configuration management if the allowed configurations are not explicitly enumerated. Instead, there is a set of objects and a set of relations between them. The user finds a configuration by expressing the desired properties and the configuration management system then tries to generate a valid configuration satisfying all the requirements.

In general, intensional methods work at higher abstraction level than extensional methods.



AND node
Selected AND node
OR node
Selected OR node

Figure 2.1: The AND/OR graph of Example 2.2.1

State-Based

In *state-based versioning* each different version of an object is explicitly stored in the version space. As a component evolves, each new version becomes an independent object and when we want to use it we do not need to know its version history.

Nearly all of the configuration systems for material products and most software configuration systems are state-based.

Change-Based

In *change-based versioning* we store only the initial version of a component explicitly. After that we store only the changes that were made at each step. Thus, we might have an object o_1 and a set of changes $\{c_1, c_2\}$ such that if we applied c_1 to o_1, we would receive object o_2 and if we then applied c_2 to o_2, we would receive o_3.

The change-based versioning is used in some software configuration management systems.

2.2 CONFIGURATION FORMALISMS

2.2.1 Graph-Based Formalisms

In a *graph-based* approach to configuration management the configuration model is presented as a graph where the nodes represent the objects and the edges represent relations between them. The article [3] gives references to many different graph-based formalisms.

AND/OR Graphs A widely used graph-based formalism is *AND/OR graphs* [46]. In an AND/OR graph the nodes are divided into AND and OR nodes. The intuition is that if an object represented by an AND node is in a configuration, all the successors of the node have also to be in it. If an OR node is in a configuration, at least one of its successors has to be in a configuration.

Example 2.2.1 Suppose that we have a program P that is composed of four parts: a, b, c, and d. Each part has two different versions, 1 and 2. The parts a and b are required to be in each product and the user may choose between parts c and d. The AND/OR graph of the program can be seen in Figure 2.1. The shaded nodes of the graph represent the objects that are chosen into a configuration. ◊

Petri Nets The article [20] introduces a *timed Petri net* based formalism for optimizing systems having both hardware and software components. The main idea is to model both types of components as transitions of the net. The places of the net correspond to the connections between hardware components and data buffers of the software components. The functions that are performed by the components are modeled using actions that are associated with transitions.

2.2.2 Rule-Based Reasoning

The first commercial product configuration systems were *rule-based* expert systems. In a rule-based system the knowledge is expressed as a set of inference rules of the form

$$\textsf{if } condition \textsf{ then } consequence. \tag{2.1}$$

In a rule-based configuration process we are given the user requirements as a set of facts and then we iterate over the set of rules adding consequences of the rules that have satisfied conditions until we cannot add anything more.

Many rule-based systems suffer from maintenance problems. If there is no formal semantics for a rule-based system, the set of valid configurations may depend on the order of the rules in the model. When there are thousands of rules, this becomes a big problem. By combining rule-based reasoning with complete logical semantics as in [41], one can have the advantages of both rule-based and model-based systems: the rules allow efficient inference of necessary components and the structure of a formal model allows easier maintenance.

2.2.3 Model-Based Reasoning

The main idea in *model-based* reasoning is that there exists a clear system model of the configuration domain and we can express it in a compact way. The individual components and their interactions are modeled using some formalism.

The model-based approaches can be divided roughly into two related classes, constraint satisfaction and logic-based approaches.

Constraint Programming In a *constraint satisfaction*-based formalism an object is defined by a set of properties and a set of ports that may be used to define connections with other objects. In addition, there is a set of *constraints* which restrict the object combinations that are allowed to exist in valid configurations.

In the standard constraint satisfaction problem we have a set of variables and we have to assign a value from a finite domain for each of them. However, there are often optional components that do not need to have a value in every configuration and we have to do unnecessary work assigning values to them. A *dynamic constraint satisfaction* formalism allows us to choose on-the-fly which variables are active in a configuration. The *generative constraint satisfaction* is a further modification of dynamic constraint satisfaction which allows specification of relations between components.

The constraint satisfaction problem has been widely researched. The references [11], [15], [24], [31], and [33] contain recent results of the area. A number of practical tools have been developed and references [14], [16], and [17] present some of them. References [39] and [38] are about dynamic constraint satisfaction problems and the articles [8] and [10] present a formalism and a tool based on generative constraint satisfaction.

Logic-based formalisms In *logic-based* approaches the configuration problem is encoded using a set of logical sentences and the valid configurations correspond to the models of the sentences. The most common approaches are based on description logics and logic programming.

There are many different variants of *description logics* in use. In a description logic we denote sets of objects by their attributes and the main inference mechanism is *subsumption*. A set C_1 subsumes the set C_2 if every instance in C_2 is included in C_1. In configuration management we model the user requirements as attributes and we want to find the corresponding set of components. The reference [48] presents a software configuration system that utilizes *feature logics* that are functional description logics. The reference [22] presents one recent description logic-based tool.

A *logic program*-based semantics combines aspects of both rule- and constraint satisfaction-based approaches. The configuration objects are modeled as atoms and the requirements and constraints are modeled as sets of inference rules. A particularly powerful semantics is the *stable model semantics* [12] which is also used in this work. The references [40] and [41] define a configuration rule language that is based on the stable model semantics.

Diagnosis approach The article [43] applies *diagnosis approach* to the reconfiguration problem. In a diagnosis system we have a system description that is divided into two parts: structural and behavioral descriptions. We are then given a set of observations about the system and we want to diagnose which components cause the observed behavior.

When we apply a diagnosis system to a configuration problem we set the observations to correspond to the user requirements. The diagnosis of the system then represents a valid configuration.

Resource-based approach In a *resource-based* approach [19] the configuration problem is formalized using producer/consumer relations. An object is modeled in terms of resources that it supplies and consumes. The goal is to find a balance between producers and consumers. For example, each electrical component may consume a fixed amount of power and there has to be enough power units in the configuration to supply them all.

2.2.4 Case-Based Reasoning

The *case-based* configuration systems are used almost exclusively in sales configuration. The configurations that have been sold to customers are stored in a knowledge base. When making a purchase, the customer first gives the requirements. The system then retrieves the configurations that satisfy most of the requirements from the knowledge base. If no existing configuration satisfies all requirements, the sales person changes one of the retrieved configurations and stores the result into the knowledge base.

Chapter 3

A Configuration Language

In this section we will present a declarative rule-based formal language RL that can be used to encode configuration problems. The language is based on a language presented in [41]. The main differences are that we will add variables and finite existential quantification, and change some notations.

The RL is defined in such way that there exists a relatively simple translation to normal logic programs with stable model semantics.

3.1 SYNTACTIC COMPONENTS

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

$$p(a_1, \ldots, a_n) \ , \tag{3.1}$$

where p is a predicate symbol and a_1 to a_n are all variables or constants. All variables are universally quantified unless otherwise stated. A *basic literal* is either an atom a or its negation not (a). Literals of the form not (a) are called negative literals. A literal is a *ground literal* if it does not have any variables.

A *truth valuation* $\mathcal{V}$ is a set of ground atoms. A valuation assigns a truth value for each literal. If an atom a is in $\mathcal{V}$, then a is true in $\mathcal{V}$, otherwise it is false. For negative literals the conditions are reversed. If an atom a is in $\mathcal{V}$, the literal not (a) is false in the valuation and vice versa.

We encode the relationships between atoms using inference *rules* of the form

$$h \leftarrow l_1, \ldots, l_n \ , \tag{3.2}$$

where the atom h is the rule *head* and the literals $l_1, \ldots, l_n$ form the rule *body*. The intuitive meaning of a rule is that if all literals in the rule body are true, the head atom has to be true also. The head atom may also be left out. In this case the rule cannot be satisfied if all body literals are true because the empty head atom cannot be true. If a rule has a head but its body is empty, it is called a *fact*. A RL *program* is a set of rules.

A valuation $\mathcal{V}$ is a *model* of the program if it satisfies all rules of the program. In particular, we are interested in *stable models* of the program. Stable models are models that satisfy an additional condition: we must have a justification for each atom that is included in the model. An atom is justified only if it occurs in a head of a rule that has its body satisfied in the model.

As basic rules encode strict implication, few elements of configuration management are difficult to express with them. The first problem comes when we want to express optionality: a component may be in a configuration if its preconditions are satisfied but we may choose to leave the component out. The second problem comes when we want to express existential quantification. For example, we might want to say that if a the functionality A is in a configuration, there should be some component that provides it in the configuration.

We can handle the optional inclusions by adding choice rules to the language. A *choice rule* is of the form

$$\{h\} \leftarrow l_1, \ldots, l_m. \tag{3.3}$$

The intuitive meaning of a choice rule is that if the body is satisfied in a model, the head atom may be added to it but we may leave it out if we want to.

We will also add to RL *existential literals* of the form

$$\exists x(l_1(x) \wedge \cdots \wedge l_n(x)) \tag{3.4}$$

where x is a variable and all $l_1(x)$ to $l_n(x)$ are basic literals. An existential literal is true in a model if there is at least some value of x that satisfies all literals $l_i(x)$. The variable is quantified over all constants of the program. We allow also negative existential literals which are of the form

$$\neg\exists x(l_1(x) \wedge \cdots \wedge l_n(x)). \tag{3.5}$$

Strictly speaking the existential literals are not necessary and we can remove them from a program by replacing an existential literal with a new atom l' and adding a new rule that defines l' to be true whenever the existential literal is satisfied. However, existential literals are a convenient form of syntactic sugar and the configuration models can be made more compact using them.

Example 3.1.1 Let R be the rule

$$a(x) \leftarrow \exists y(p(x,y) \wedge \mathrm{not}\, r(x,y)), \neg\exists z(p(z,x) \wedge \neg r(z,x)), b(x) \tag{3.6}$$

We can replace the two existential literals with two new literals $l_1(x)$ and $\mathrm{not}\, l_2(x)$ and the corresponding rules:

$$\begin{aligned} a(x) &\leftarrow l_1(x), \mathrm{not}\, l_2(x), b(x) \\ l_1(x) &\leftarrow p(x,y), \mathrm{not}\, r(x,y) \\ l_2(x) &\leftarrow p(z,x), \mathrm{not}\, r(z,x) \end{aligned} \tag{3.7}$$

$\Diamond$

3.2 SEMANTICS

We will define the formal semantics of RL to in a way similar to the stable model semantics for general logic programs [12]. The intuition of the semantics is that a stable model of a program may be seen as a set of rational beliefs about the program. That is, if we think that a program is a knowledge base encoding the relationships between objects and a stable model is the set of things in our universe that we believe are true, then our beliefs have to be consistent and well-founded. Consistency means that we do not believe in two contradictory things and well-foundedness means that we have to have some reason for our belief. In addition, the semantics has implicit negation: if we cannot deduce some atom, we assume that its negation is true.

For example, suppose we would like to express facts about birds using the following rules

$$flies(X) \leftarrow bird(X), \mathrm{not}\, flightless(X)$$
$$bird(tweety) \leftarrow .$$

The first rule says that a bird can fly if it is not flightless and the second rule asserts that Tweety is a bird. We can now infer that Tweety can fly since it is a bird and we do not have any reason to suppose that it is flightless. The program has exactly one stable model, $\{bird(tweety), flies(tweety)\}$.

Suppose that we add the rules

$$flightless(X) \leftarrow penguin(X)$$
$$penguin(tweety) \leftarrow$$

to the knowledge base. Now we can deduce that because Tweety is a penguin, it cannot fly and thus the only stable model of the extended program is

$$\{bird(tweety), penguin(tweety), flightless(tweety)\} \ .$$

As adding more rules to the program removed the atom $flies(tweety)$ from the model, we see that the stable model semantics is non-monotonic.

When we want to find whether a set M of atoms of a program P is a stable model, we do it in three steps:

1. We construct a ground program P_g from P by instantiating all rules with all possible variable bindings.

2. We construct a reduct P_g^M of P_g with the respect to the set M. The idea of the reduct is to remove all negative literals and choice rules from the program so that we can we can straightforwardly compute the deductive closure of the program.

3. We check whether M is equal to the minimal model of P_g^M. If it is, M is a stable model, otherwise it is not.

Instantiation

The *Herbrand universe* $\mathbf{HU}(P)$ of a program P is the set of all constants that are in P. An *instance* of a literal or a rule is constructed by replacing all variables in it with constants. The *Herbrand instantiation* $\mathbf{HI}(P)$ is the set of all ground instances of rules in P that can be constructed using terms in $\mathbf{HU}(P)$. For a RL program P we define the program P_g to be the Herbrand instantiation $\mathbf{HI}(P)$.

Example 3.2.1 Let the program P consist of rules

$$
\begin{aligned}
a(x) \leftarrow b(x), \neg\exists y(p(x,y))\\
b(1) \leftarrow\\
b(2) \leftarrow\\
p(1,2) \leftarrow
\end{aligned}
\tag{3.8}
$$

We remove the existential literal with the following rules

$$
\begin{aligned}
a(x) \leftarrow & b(x), \operatorname{not} l(x)\\
l(x) \leftarrow & p(x,y)
\end{aligned}
\tag{3.9}
$$

The instantiated program P_g consists of rules

$$
\begin{array}{ccc}
a(1) \leftarrow b(1), \operatorname{not} l(1) & ; & a(2) \leftarrow b(2), \operatorname{not} l(2)\\
l(1) \leftarrow p(1,1) & ; & l(1) \leftarrow p(1,2)\\
l(2) \leftarrow p(2,1) & ; & l(2) \leftarrow p(2,2)\\
b(1) \leftarrow & ; & b(2) \leftarrow\\
p(1,2) \leftarrow & &
\end{array}
\tag{3.10}
$$

$\Diamond$

Stable Models

We construct the reduct P_g^M of P_g with respect to M by

1. removing each rule that has a negative literal not (a) in its body with $a \in M$;
2. removing each rule that has $\{a\}$ as its head when $a \notin M$;
3. removing all negative literals from the bodies of remaining rules and all braces from the choice rule heads.

As P_g^M is negation-free, it has a unique minimal model. If this model coincides with M, then M is a *stable model* of P. A program may have one, none, or many stable models.

Example 3.2.2 Let P be the program

$$
\begin{aligned}
a \leftarrow \operatorname{not} b\\
b \leftarrow \operatorname{not} a
\end{aligned}
\tag{3.11}
$$

Now P has two stable models: $\{a\}$ and $\{b\}$. To see that the first one is a stable model we construct the reduct $P^{\{a\}} = \{a \leftarrow\}$. The unique minimal model of the reduct is $\{a\}$, which coincides with the model. On the other hand, $M = \{a, b\}$ is a model but it is not stable since its reduct is empty. $\Diamond$

3.3 OPTIMIZATION

Often we would like to find configurations that are optimal with respect to some criteria. We handle them by adding a new type of statement to the language.

A *maximize statement* is of the form

$$\mathsf{maximize}\ \{l_1, \ldots, l_n\}\ , \tag{3.12}$$

where $l_1, \ldots, l_n$ are literals. We can use a maximize statement to express the fact that we would like to have as many of the l_i in the configuration as possible. If there are more than one maximize statement, they are considered in fixed order. When we compare two configurations, we initially examine only the first maximize statement. Only if both configurations have the same weight, the next maximize statement is considered.

3.4 COMPUTATIONAL COMPLEXITY

Theorem 3.4.1 *The problem of finding a stable model for a ground* RL *program is* NP*-complete.*

Proof. As the problem of finding a stable model of a normal logic program is NP-complete [21] and all normal programs are also RL programs, the corresponding problem for RL is automatically NP-hard.

To prove the completeness we show how a RL program P can be reduced into a normal logic program in polynomial time.

First, we replace all choice rules of the form

$$\{h\} \leftarrow l_1, \ldots, l_m. \tag{3.13}$$

with two new rules

$$\begin{aligned} h \leftarrow & l_1, \ldots, l_m, \mathrm{not}\, h' \\ h' \leftarrow & \mathrm{not}\, h \end{aligned} \tag{3.14}$$

where h' is a new atom. Now we can infer h only if $l_1, \ldots, l_m$ are true and h' is not true and h' is true exactly when h is not true. Thus, we have introduced an explicit choice point to the program in place of the implicit one defined by the choice rule.

We replace an existential literal with the construct specified in Section 3.1. This reduction can be done in a polynomial time with the respect to the number of literals in the program since each replacement only adds one literal and one rule. □

From the above theorem and the definition of the complexity class NP we immediately get the following corollary.

Corollary 3.4.1 *Given a set M and a ground* RL *program P of atoms we can verify whether M is a stable model of P in polynomial time.*

We will implement the rule language RL by translating the programs to normal logic programs with stable model semantics. We do this because there is an existing efficient implementation, `smodels`[1] [25, 34, 26, 35], for normal logic programs. As `smodels` works only for variable-free programs, we have to use a front-end to ground the programs. We will use `lparse`[2] [45] which has been developed by the author.

If a RL program does not use choice rules or existential literals, it is automatically a normal logic program. Fortunately, `smodels` has direct support for choice rules and maximisation [36],[27] and we only have to consider existential literals. We can model them using `smodels` constraint rules and conditional literals. The constraint rules are unrelated to the constraints imposed by a configuration model. Unless otherwise noted, in this work the term constraint is used to denote the configuration constraints.

A `smodels` *constraint rule* is of the form

$$h \leftarrow L \leq \{l_1, \ldots, l_n\} \leq U \ , \tag{3.15}$$

where L and U are integer *lower* and *upper* bounds. Intuitively, we can infer that the head atom h is true if the number of satisfied literals l_i is between the lower and upper bounds.

Example 3.5.1 Given the constraint rule

$$e \leftarrow 2 \leq \{a, b, c, d\} \leq 3 \ , \tag{3.16}$$

we can infer that e is true whenever two or three of the atoms a, b, c, and d are true. ◊

A *conditional literal* of the form

$$a(x) : d(x) \tag{3.17}$$

represents the set of atoms $\{a(x) \mid d(x) \in M\}$. That is, the predicate d is used to give all possible values of x for $a(x)$.

Example 3.5.2 Suppose that a program P contains the following rules for the predicate d:

$$\begin{aligned} d(1) &\leftarrow \\ d(2) &\leftarrow \\ d(3) &\leftarrow \end{aligned} \tag{3.18}$$

Now the conditional literal $a(x) : d(x)$ represents the set $\{a(1), a(2), a(3)\}$ ◊

We can now model a positive existential literal $\exists x(P(x) \wedge Q(x))$ with a `smodels` constraint of the form

$$1 \leq \{p(X) : q(X)\} \ . \tag{3.19}$$

[1]The `smodels` is available at <`url:http://www.tcs.hut.fi/pub/smodels`>

[2]The `lparse` is available at <`url:http://www.tcs.hut.fi/pub/smodels/lparse`>

An negative existential literal $\neg\exists x(P(x) \land Q(x))$ can be encoded with

$$\{p(X) : q(X)\} \leq 0. \tag{3.20}$$

The two `smodels` constraints give only the general form of the translation since each existential literal has to be translated independently. The main problem is that `lparse` accepts only a non-recursively defined predicate as a right argument of a conditional literal. Therefore, at most one predicate in an existential literal may be recursively defined. If there are more recursively defined predicates, we have to remove the literal as in Example 3.1.1.

Chapter 4

The Debian GNU/Linux System

In this section we describe the Debian GNU/Linux system in detail. We introduce the Debian packages and their relationships in Section 4.2. In Section 4.3 we see the current state of the configuration management software. Section 4.4 presents a number of problems with the current approach and propose a few improvements to it.

4.1 A DEBIAN CONFIGURATION

A Debian configuration is a set of software packages that are installed on the user's computer. Not all package combinations are legal configurations since a package might depend on another one or two packages may be mutually exclusive.

Usually the configurations are constructed using the `dselect` tool that allows the user to select the interesting packages. After each change the `dselect` checks whether the resulting configuration is legal and if it is not it displays an error message.

4.2 DEBIAN PACKAGES

Currently there are two types of packages in Debian system, concrete and virtual packages. A *concrete package* is composed of a set of files that provide some functionality. A concrete package may provide a program, a library, or some document files. A *virtual package* represents an abstract concept that can be provided by many different concrete packages.

Example 4.2.1 The virtual package mail-reader is used to express the fact that there is a program which can be used to read mail in the configuration. The package mail-reader is provided by concrete packages af, mailx, mh, mutt, nmh, tkmail, and vm. If any one of these packages is in the configuration, mail-reader is also in it. ◊

In addition to these two package types, we define a new type, namely software packages. A *software package* is an abstraction that represents some specific software item that can be provided by a concrete package. For example, a concrete package gawk-3.0.3 provides version 3.0.3 of software package

Table 4.1: The attributes of Debian packages

Attribute	Meaning
Name	Name of the software package
Version	Version of the software package
Filename	Name of the concrete package including its location in the distribution directory hierarchy. The actual filename is composed from package name and version number.
Priority	How important the package is for the system. Possible values are `required`, `important`, `standard`, `optional`, and `extra`.
Section	The basic classification of the package. Possible values include `admin`, `base`, and `text` among others
Size	Size of the compressed package
Installed-size	Size of the program after it has been installed.
Architecture	The computer architecture that the package was compiled for. Possible values are currently `i386`, `alpha`, `sparc`, and `m689`.
Maintainer	The name of the person who is responsible for the package
Description	A short summary of the purpose and use of the package.
Checksum	A hash value calculated with MD5-algorithm that can be used to check that nobody has been tampering with the package.

`gawk`.

The notion of software packages is not used in Debian documentation but it is highly useful to be able to make a distinction between virtual and software packages. The difference between the two types is that there may be many different packages that provide a virtual package in a single configuration but there may be at most one concrete version of a software package in one configuration.

We will use general term *package* to stand for all package types.

There is a set of attributes defined for each concrete package[1]. The attributes are presented in Table 4.1.

Not all attributes are important for a configuration task. The most important ones are

- name and version, since we must be able to tell the packages apart; and
- priority, as we need to know how important a package is.

In addition, the package size may sometimes be an important factor when deciding whether to include that package or not. However, we will not consider that aspect in this work.

The Debian system packages are described in detail in [6].

[1] Currently there is no way to express attributes of virtual packages in the Debian system.

4.2.1 Package Versions

Every concrete package has a version string assigned to it. The version space is assumed to be linear, that is, a new version of a package always supersedes the old versions. If the version history of a package branches, a new software package has to be created for the other branch. The version string is of the form

[epoch:]upstream-version[-debian-revision]

where the parts in brackets are optional.

The version numbers are compared from left to right. All numeric parts are compared by value and non-numeric parts are ordered lexicographically.

The epoch part is used only to recover the model when there are errors in the version numbering scheme and the comparison algorithm would give wrong results. The upstream-version designates the main part of the version and it is assigned by the author of the program. The last part of the string is assigned by the Debian maintainer of the package to keep track of Debian-specific modifications.

4.2.2 Priority Classes

The package priority is the most important attribute after name and version. There are five distinct priority classes.

1. *Required* packages are necessary for the system to function correctly. All tools that are needed to install the system or fix defects in it are in this category.

2. *Important* packages include utilities that should be found in almost all Unix-like systems. If only the *important* and *required* packages are installed, the system will be usable but severely limited.

3. *Standard* packages provide most of the utilities that users expect to find on a Linux system. By default, all *standard* packages are included in system unless otherwise directed.

4. *Optional* packages provide applications and utilities that are useful but that may also be left out without compromising the usability of the system. Most of the large applications are *optional*.

5. *Extra* packages are in conflict with others of higher priority but they provide a special functionality that may be useful in some cases.

4.2.3 Package Relationships

Packages do not exist in isolation, as they interact with other packages. A package might not function if another one is missing or two packages might both offer the same functionality. Debian defines seven possible relationships between packages: dependency, conflict, recommendation, suggestion, pre-dependency, replacement, and providing.

Except for 'providing', each relation may also be parametrized by a version clause. A version clause restricts the versions of the software package for which the relation holds. The allowed version relations are $<, \leq, =, \geq$, and, $>$.

Example 4.2.2 Given a relation definition

$$\mathsf{A} \text{ depends on } \mathsf{B}\ (\geq 1.2)\ , \tag{4.1}$$

we cannot have A in a configuration if we do not have version 1.2 or later of B in it. ◊

The relations are defined in following way:

1. Dependency: A package A *depends* on a package B if A cannot be used at all if B is not installed.

2. Conflict: A package A *conflicts* with a package B if A will not operate if B is installed on the system. Ideally this relation would be symmetric but it is possible that the maintainer of the older package is not aware that a later package conflicts with it.

3. Recommendation: A package A *recommends* a package B if the package maintainer judges that most users would not want A without functionality provided by B.

4. Suggestion: A package A *suggests* a package B if B enhances the functionality of A.

5. Replacement: A package A *replaces* a package B if the files installed by B will be removed or overwritten by files in A. In practice, replacement is only used to guide the configuration process. If two packages are in conflict and one of them replaces the other, the replacing one is preferred.

6. Providing: A package A *provides* a package B when A incorporates all files and functionality of B.

7. Pre-dependency: A package A *pre-depends* on a package B when A cannot be installed if B is not completely installed on the system. Pre-dependency relation is normally used only for packages where correct installation order is critical. With normal dependency, a package may be unpacked even if its dependencies are not satisfied, but a package with unsatisfied pre-dependencies cannot be even unpacked.

The relations that issue a kind of dependency (depends, recommends, suggests and pre-depends) may define also a set of alternative packages. In that case, at least one of the packages has to be in the configuration.

Example 4.2.3 Given a relation definition

$$\mathsf{A} \text{ depends on } \mathsf{B}\ (\geq 1.2) \text{ or } \mathsf{C}\ (\leq 2.5), \tag{4.2}$$

we must have either B version 1.2 or later, or C version 2.5 or earlier if we want to have A in the configuration. ◊

```
function configure-package-set-try-1(PackageList lst)
    changed := true
    while lst ≠ ∅ ∧ changed = true do
        changed := false
        foreach package pck ∈ lst do
            if satisfied-depends(pck) do
                configure(pck)
                lst.remove(pck)
                changed := true
            end if
        end for
    end while
    if changed = true do
        return 'done
    else
        return 'unresolved-dependencies
    end if
end
```

Figure 4.1: The `dpkg` algorithm for the first round of dependency resolution.

4.2.4 Virtual Packages

In Debian 2.1 virtual packages are defined only implicitly. If a concrete package provides a package, that package is a virtual package. As there is no general way of defining a virtual package, it is not possible to define relations between two virtual packages.

Virtual packages do not have versions attached to them. This is because Debian does not make the distinction between virtual and software packages. There may be a concrete package and a virtual package with the same name and Debian disambiguates the references between them by the existence of a version number. If a package reference has a version number attached to it, it is determined to be a concrete package, otherwise it is a virtual package.

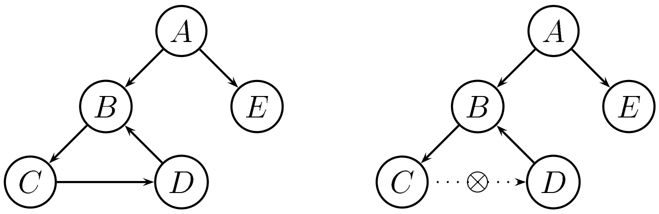
4.3 CURRENT STATUS

The relations between packages are defined in a huge package description file that has an entry for all concrete packages. The resolution of dependencies and conflicts is done using the `dpkg` program.

Concrete packages are installed in two steps. First the packages are all unpacked and their files are placed in correct places of the directory hierarchy. Then `dpkg` goes through the list of packages to install and tries to configure them in correct order, so that a package is configured only after all its dependencies are configured. The `dpkg` does not construct a full model of the configuration at any time but it works a package at a time trying to meet its requirements.

```
function configure-package-set-try-2(PackageList lst)
    changed := true
    while lst ≠ ∅ ∧ changed = true do
        had_cycle := try-break-cycle(lst)
        if had_cycle = true do
            changed := true
        else
            changed := false
        end if
        foreach package pck ∈ lst do
            if satisfied-depends(pck) do
                configure(pck)
                lst.remove(pck)
                changed := true
            end if
        end for
    end while
    if changed = true do
        return 'done
    else
        return 'unresolved-dependencies
    end if
end
```

Figure 4.2: The `dpkg` algorithm for the second round of dependency resolution.



(a) Before (b) After

Figure 4.3: How `dpkg` resolves a dependency cycle

4.3.1 Dependencies

According to the source code of `dpkg`, the dependency resolution and configuration algorithm works roughly in three steps:

1. The list of packages selected to be installed is walked through and all packages that have all their dependencies satisfied are configured and removed from the list. This is repeated until after one iteration we cannot remove any packages from the list. If the list is empty at that point, we have configured all packages and can stop. Otherwise, we will move on to the second step. The pseudo-code version of this algorithm is presented in Figure 4.1.

2. We scan through the package list and try to find a cycle in the dependency graph of the packages. If a cycle is found, we break it by removing one dependency link. After that we proceed like we did in the first step. If at some point there are no more cycles left and there still are packages with unsatisfied dependencies, we move on to the third step. This algorithm is presented in Figure 4.2.

3. We check if the user wants to force the configuration regardless of unsatisfied dependencies. If this happens, we configure the remaining packages, otherwise we stop and inform the user about unsatisfied dependencies.

The `dpkg` does not try to add new packages to the package list to resolve any missing dependencies. Instead, it expects that the user either explicitly gives all relevant packages or uses a front-end that performs the calculation. The default front-end is `dselect` which is introduced in Section 4.3.3.

As `dpkg` does not fully utilize the knowledge about the configuration during installation process, it has to enforce the dependency constraints at runtime. That is, it refuses to configure a package with unsatisfied dependencies even if the dependencies will be satisfied later in the configuration run.

Example 4.3.1 Suppose that the user wants to add a package A to a configuration and the package depends on packages B, C, D, and E as pictured in Figure 4.3.

Initially, `dpkg` walks through the package list until it finds a package with all dependencies satisfied. The only such package is E which is now configured and removed from the list. As the dependencies of other packages cannot be satisfied, `dpkg` moves into step 2.

Scanning through the package list `dpkg` finds the cycle

$$B \to C \to D \to B \tag{4.3}$$

in the graph. The cycle is then broken by removing one edge from the graph. In this case, the edge $C \to D$ was removed. Now `dpkg` can satisfy all remaining dependencies by configuring first C, then B, D, and finally A, in that order. ◊

4.3.2 Conflicts

The `dpkg` conflict handling is simpler than the dependency handling. If `dpkg` finds out that a package it is trying to install causes a conflict, it checks whether the other package can be removed without violating dependencies of any installed package. If there are no problems, the other package is automatically removed and the installation proceeds. Otherwise, `dpkg` signals error and stops. The user may force `dpkg` to install conflicting packages by giving a correct command-line argument.

4.3.3 Dselect

While `dpkg` does the actual work of maintaining the configuration, it needs to have complete lists of packages that need to be installed or removed. The primary user interface for it is the `dselect` program. The `dselect` allows the user to select packages for installation and removal and after each selection it calculates whether the change causes a dependency problem or a conflict.

If a conflict is detected, the conflicting packages are shown to the user in addition to currently installed packages that depend on either of them. The user has to choose which package to retain.

If a missing dependency is detected, `dselect` presents the problem to the user and suggests a package that provides the required functionality. If there are more than one package that can satisfy the dependency the choice of suggestion is made according to a set of preference rules which are explained in Section 4.3.4.

After the user has made all desired changes, `dselect` calls `dpkg` to perform the actual updating of the system configuration.

4.3.4 Preference Rules

The Debian Packaging manual [6] introduces three preference rules that are used to select a package to satisfy a dependency:

1. If the alternative packages have different priorities, the one with highest

priority is preferred. For example, a package with priority standard is preferred over extra packages.

2. If the packages have same priority, the one that is listed first in the relation definition field of the package description file is preferred. For example, suppose that the package description file includes lines:

   ```
   Package: af
   Depends: smail | mail-transport-agent
   ```

   If we want to add af to the configuration, we must satisfy the dependency. Because smail is the first in package in the line, it will be preferred over mail-transport-agent.

3. Otherwise, the choice is essentially random.

4.4 PROBLEMS WITH THE CURRENT PRACTICE

While dpkg usually works correctly, there are still some problems with the Debian configuration system.

One of the problems is that sometimes dselect and dpkg have differing opinions about allowed configurations. It is possible to have a situation where dselect determines that a configuration is legal, but dpkg cannot deduce how to reach that configuration. For example, the author once had problems because dpkg did not allow to change the installed C++-compiler from g++ to egcs. Before egcs could be installed, g++ had to be removed, but removing it would have violated many dependencies. At that occasion dpkg did not realize that installing egcs right afterwards would have satisfied those dependencies, and it did not allow g++ to be removed.

Another problem occurs because there is no way of generating aggregated packages. Many packages, especially ones providing graphics software, depend on a large number of packages and maintaining the relation definitions may prove difficult when the packages evolve. In addition, many of the packages have same dependencies. If we could define a new type of virtual packages that represents a collection of concrete packages, we could use them to simplify the maintenance task.

Example 4.4.1 Most of the graphics packages need libraries for handling different image formats and display types. These libraries are same for all graphics programs. Currently, the description files for these packages look much like

```
Package: arena
Depends: libc6, libjpegg6a, libpng0g, xlib6g (>= 3.3-5),
         xpm4g (>= 3.4j-0), zlib1g
```

If we could define an aggregated package image-formats, we could simplify this entry to

```
Package: arena
Depends: libc6, image-formats, xlib6g (>= 3.3-5), zlib1g
```

and add a definition

```
Package: image-formats
Contains: libjpegg6a, libpng0g, xpm4g (>= 3.4j-0)
```

to the description file. ◊

In some cases it could be useful if we could have more fine-grained control over relations. For example, now a package either always provides a virtual package or it does not provide it ever, while in some cases it we would like to use two small packages together to provide functionality of a single virtual package.

Example 4.4.2 Suppose we have a virtual package A and concrete packages B and C. When both packages are installed they provide the functionality of A but neither of them implements it completely.

We would like to infer that A is in the configuration if both B and C are, so we need a method of expressing the following two statements:

B provides A if C is also in the configuration
C provides A if B is also in the configuration

◊

Finally, there is no formal semantics defined for the package relations. Currently, the semantics is defined by the behavior of `dpkg`. While the system works reasonably well, the lack of formal definition leads to certain maintenance problem as a package maintainer has to know how `dpkg` works to be certain that all relations are correctly defined.

Example 4.4.3 The package awe-drv is used to provide support for SoundBlaster AWE - type of sound cards. The package description file includes the following lines for it:

```
Package: awe-drv
Conflicts: awe-utils, awe-patch
Provides: awe-utils, awe-patch
Replaces: awe-utils, awe-patch
```

In effect, the package at the same time seems to provide two virtual packages, replace the same packages, and even be in conflict with them.

This apparent contradiction can be settled if we examine the way how `dselect` and `dpkg` process this package. When a package conflicts with a virtual package that it provides itself, `dpkg` interprets it as meaning that when the package is in the configuration it should be the only package providing the functionality.

The final relation declaration is actually an error and it will be ignored by `dpkg`. The packaging manual [6] states that a package may replace only packages that are mentioned with their real names, and virtual ones are not considered. ◊

4.4.1 Proposed Improvements

In this work we will strive to augment the Debian configuration system by adding a formal semantics, aggregated packages, software packages, and conditional relations. The resulting configuration model will be formally defined in Section 5.

Formal semantics We will define a formal semantics for Debian GNU/Linux configuration management using the configuration rule language RL defined in Chapter 3.

Aggregated packages We will include the concept of aggregated packages into the model. An aggregated package is composed of other packages. If all components of an aggregated package are in a configuration, the aggregation is also in it.

Conditional relations We will define a propositional language that allows us to specify when a relation holds and when it does not. The propositional atoms of the language will be the packages of the configuration model and a proposition asserts that the relation holds when some combination of packages is in the configuration.

Relations for virtual packages Our rule language allows us to express relationships between virtual and aggregate packages in addition to the current way of defining relations only for concrete packages.

Chapter 5

Debian GNU/Linux Configuration

In this chapter we will define a formal model that captures the existing practice of version 2.1 of the Debian GNU/Linux system as described in Chapter 4 and improvements proposed in Section 4.4.1. We have three main goals for this formulation:

1. the model should be sound and complete;
2. the model should be computationally feasible; and
3. it should be possible to generate the model automatically from existing Debian configuration information.

We will first try to identify the important parts of Debian configuration problem and then construct a model using the rule language developed in Chapter 3. As the full model will be quite large, only the basic construction is presented here and the details are presented in Chapter 7.

5.1 THE BASIC IDEA

A Debian configuration is essentially a set of installed packages. Thus, in our configuration model will include a set S of available packages and a configuration c is a subset of S. In terms of the RL language, we will model the packages with constants and the relations between them are modeled with predicates and inference rules. A configuration should be minimal in the sense that every package in it should have some reason why it is included. For example, if the user selected a package, that package is justified in the configuration.

We will use the predicate

$$\mathsf{in(P)} \tag{5.1}$$

to denote that a package P is in a configuration. We will take the approach that if a package is justified, it may be in a configuration. However, a justified package may also be left out if it does not cause any problems. We can capture this idea with the choice rule

$$\{\mathsf{in(P)}\} \leftarrow \mathsf{justified(P)}. \tag{5.2}$$

In addition, we will define a set of constraints that restrict the number of valid configurations. For example, we want to leave out configurations that have two conflicting packages. This may be done using a rule of the form

$$\leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2). \tag{5.3}$$

Finally, we need a way to express preferences on different configurations. We take the approach that the preferences are expressed using optimization functions. Each optimization function assigns a cost value to every configuration, and the ones with least cost are preferred.

5.2 PACKAGES

As we saw in Section 4.2, there are two different types of packages, concrete and virtual, in the Debian configuration system and we would like to add software and aggregated packages. Therefore, we need a set of predicates that assign a type to a constant representing a package.

We will use the following predicates:

$$\begin{array}{ccc} \mathtt{concrete(C)} & ; & \mathtt{virtual(V)} \\ \mathtt{aggregate(A)} & ; & \mathtt{software(S)} \end{array} \tag{5.4}$$

In addition, we want to have a predicate for expressing the general concept of packages so we define a new predicate `package(P)`, and add the following rule for each different package type

$$\mathsf{package}(\mathsf{C}) \leftarrow \mathsf{concrete}(\mathsf{C}). \tag{5.5}$$

We still need a way to associate the concrete packages to the corresponding software packages since otherwise we could not model the version space of the configuration model. We will define a new predicate

$$\mathtt{available\_version(S, C)} \tag{5.6}$$

to denote the fact that concrete `C` is a version of `S`. We can use a similar construct to add versioning for virtual packages.

Each package has a priority assigned to it. The priorities are modeled using the following predicates

$$\begin{array}{ccc} \mathtt{required(P)} & ; & \mathtt{important(P)} \\ \mathtt{standard(P)} & ; & \mathtt{optional(P)} \\ \mathtt{extra(P)} & & \end{array} \tag{5.7}$$

Later, in Section 5.7 we need to be able to compare the priority levels numerically. We will assign the values from one to five to the levels, with the lowest priority `extra` receiving the value 1 and the highest priority `required` receiving the value 5.

Name	Purpose
depends(X,Y)	X will not function without Y
conflicts(X, Y)	X will not function when Y is present
recommends(X, Y)	Y should be in a configuration if X is present
suggests(X, Y)	Y enhances the functionality of X
provides(X, Y)	X provides the virtual package Y
pre-depends(X, Y)	X cannot be installed if Y is not installed
contains(X, Y)	X is an aggregate and it contains Y

Table 5.1: Debian relation types

5.3 RELATIONS

In Section 4.2.3 we identified a set of relationships between packages. We will model each of them with a predicate of the form

$$\texttt{depends}(P_1,\ P_2) \tag{5.8}$$

where P_1 and P_2 are any packages. The relations are presented in Table 5.1.

The simple predicates of the form (5.8) are not enough if we want to express version clauses and other conditional relations. However, throughout this section we will use those predicates as a convenient abstraction. If an atom depends(P_1, P_2) is true in a stable model it should be interpreted as "the package P_1 depends on the package P_2 *in this configuration*." The conditional relations are discussed in detail later in Section 5.9

As the replacing relation is only used to guide the `dpkg` configuration process, it is left out of this model.

5.4 USER REQUIREMENTS

There are two main types of user requirements. The user may either want to have a package that otherwise would be left out or she may want to leave a package out that would otherwise be included.

We will model the first kind of user requirements with a predicate

$$\texttt{user\_include}(P). \tag{5.9}$$

An atom user_include(P) is true in a model when the user explicitly selected the package P to be in the configuration.

The other kind of user requirements is modeled using a complementary predicate

$$\texttt{user\_exclude}(P). \tag{5.10}$$

An atom user_exclude(P) is true in a model when the user explicitly removed the package P from the configuration.

5.5 CONSTRAINTS

The Debian system includes a number of constraints that a configuration has to satisfy. The two most important ones are dependency and conflict. A package may not be in a configuration if some of its dependencies are unsatisfied and two conflicting packages may not be in a configuration. Additionally, there are few other constraints. For example, a virtual package may not be in a configuration if no one provides it and a required package may not be left out.

The Debian GNU/Linux configuration model has the following seven constraints:

Dependency

A package P_1 may not be in a configuration if there exists a package P_2 such that P_1 depends on P_2 and P_2 is not in the configuration.

$$\leftarrow \mathsf{in}(\mathsf{P}_1), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{not\ in}(\mathsf{P}_2) \tag{5.11}$$

Conflict

If two packages P_1 and P_2 conflict with each other, it is not possible that both are in same configuration.

$$\leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2) \tag{5.12}$$

Providing

A virtual package V may not be in a configuration if it is not provided by some concrete package C that is in the configuration.

$$\leftarrow \mathsf{in}(\mathsf{V}), \neg\exists \mathsf{C}(\mathsf{provides}(\mathsf{C}, \mathsf{V}) \wedge \mathsf{in}(\mathsf{C})) \tag{5.13}$$

Conversely, if a concrete package is in a configuration, all virtual packages it provides have to be in it also.

$$\leftarrow \mathsf{in}(\mathsf{C}), \mathsf{provides}(\mathsf{C}, \mathsf{V}), \mathsf{not\ in}(\mathsf{V})) \tag{5.14}$$

Concrete Versions

If a software package S is in a configuration, there has to be a concrete version C of it in the configuration.

$$\leftarrow \mathsf{in}(\mathsf{S}), \neg\exists \mathsf{C}(\mathsf{available\_version}(\mathsf{S}, \mathsf{C}) \wedge \mathsf{in}(\mathsf{C})) \tag{5.15}$$

Likewise, if a concrete version of a package is in a configuration, the corresponding software package has to be in it.

$$\leftarrow \mathsf{in}(\mathsf{C}), \mathsf{available\_version}(\mathsf{S}, \mathsf{C}), \mathsf{not\ in}(\mathsf{S}) \tag{5.16}$$

There may be only one concrete version of a software package installed at a time.

$$\begin{aligned} \leftarrow\, & \mathsf{in}(\mathsf{C}_1), \\ & \mathsf{available\_version}(\mathsf{S}, \mathsf{C}_1), \\ & \exists \mathsf{C}_2(\mathsf{available\_version}(\mathsf{S}, \mathsf{C}_2) \wedge \mathsf{in}(\mathsf{C}_2) \wedge \mathsf{C}_1 \neq \mathsf{C}_2) \end{aligned} \tag{5.17}$$

Required Packages

If a package has required priority, it has to be included in a configuration.

$$\leftarrow \mathsf{required}(\mathsf{P}), \mathsf{not\ in}(\mathsf{P}) \tag{5.18}$$

User Requirements

If the selected a package P to be in a configuration, it has to be included. Likewise, if he excluded a package it must be left out.

$$\begin{aligned} &\leftarrow \mathsf{user\_include}(\mathsf{P}), \mathsf{not\ in}(\mathsf{P}) \\ &\leftarrow \mathsf{user\_exclude}(\mathsf{P}), \mathsf{in}(\mathsf{P}) \end{aligned} \tag{5.19}$$

Aggregations

If an aggregated package A is in a configuration, all its components have to be in it.

$$\leftarrow \mathsf{in}(\mathsf{A}), \exists \mathsf{P}(\mathsf{contains}(\mathsf{A}, \mathsf{P}) \land \mathsf{not\ in}(\mathsf{P})) \tag{5.20}$$

5.6 JUSTIFICATIONS

A configuration is a subset of packages that are justified. Thus, a package may be in a configuration only if it is justified.

$$\{\mathsf{in}(\mathsf{P})\} \leftarrow \mathsf{justified}(\mathsf{P}) \tag{5.21}$$

There are three basic ways how a package may become justified:

1. all required, important, and standard packages are justified;
2. all user-selected packages are justified; and
3. a package that is in the configuration depends on or recommends the package.

The following justification rules are defined:

1. A package is justified if its priority is standard or higher.

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{standard}(\mathsf{P}) \tag{5.22}$$

   We need also corresponding rules for required and important packages.

2. All packages selected by the user are justified

$$\mathsf{justified}(\mathsf{P}) \leftarrow \mathsf{user\_include}(\mathsf{P}) \tag{5.23}$$

3. If an included package P_1 recommends, suggests, or depends on a package P_2, P_2 is justified.

$$\mathsf{justified}(\mathsf{P}_2) \leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{5.24}$$

   We need also corresponding rules for recommendations and suggestions.

4. A concrete package C is justified if it provides a virtual package V that is in the configuration.

$$\mathsf{justified}(\mathsf{C}) \leftarrow \mathsf{in}(\mathsf{V}), \mathsf{provides}(\mathsf{C}, \mathsf{V}) \tag{5.25}$$

5. A virtual package V is justified if a concrete package that provides it is in the configuration.

$$\mathsf{justified}(\mathsf{V}) \leftarrow \mathsf{in}(\mathsf{C}), \mathsf{provides}(\mathsf{C}, \mathsf{V}) \tag{5.26}$$

6. If a software package S is in a configuration, all concrete versions of it are justified.

$$\mathsf{justified}(\mathsf{C}) \leftarrow \mathsf{in}(\mathsf{S}), \mathsf{available\_version}(\mathsf{S}, \mathsf{C}) \tag{5.27}$$

7. Conversely, if a concrete package C is in the configuration, the corresponding software package S is justified.

$$\mathsf{justified}(\mathsf{S}) \leftarrow \mathsf{in}(\mathsf{C}), \mathsf{available\_version}(\mathsf{S}, \mathsf{C}) \tag{5.28}$$

5.7 OPTIMIZATION FUNCTIONS

Usually there many configurations that satisfy all user requirements and we have to choose one among them. We saw few preference rules in Section 4.3.4 that are currently used to select packages. The preference rules are not complete as there are situations where `dpkg` makes essentially random selections.

We will take the approach that we define a set of optimization functions. Each function assigns a weight to a configuration and we will prefer configurations with large weights over small weights. Naturally, there are many ways to define weight functions and different users have different preferences. Below we see only one method of defining the optimization functions.

We define the following weight functions:

Priority

We want to prefer packages with high priority over packages with low priority. We can do this by defining a weight function w_p as

$$w_p(c) = \sum_{i=1}^{5} n^i \cdot n_{ci} \tag{5.29}$$

where c is a configuration, n is the total number of packages, and n_{ci} is the number of packages of priority level i in the configuration c.

By this definition, we will prefer one important package over any number of standard or optional packages and a standard package over any number of optional ones.

Recommendations
We want to have as many recommended packages in the configuration as possible. For this, we define a weight function w_r as follows:

$$w_r(c) = \sum_{p \text{ is a package}} w_{cr}(p) \tag{5.30}$$

where $w_{cr}(p)$ is a function

$$w_{cr}(p) = \begin{cases} 1, & \mathsf{in}(p) \wedge \exists p'(\mathsf{recommends}(p', p) \wedge \mathsf{in}(p')) \\ 0, & \text{otherwise} \end{cases} \tag{5.31}$$

Suggestions We want to have as many suggested packages in the configuration as possible. For this, we define a weight function w_s as follows:

$$w_s(c) = \sum_{p \text{ is a package}} w_{cs}(p) \tag{5.32}$$

where $w_{cs}(p)$ is a function

$$w_{cs}(p) = \begin{cases} 1, & \mathsf{in}(p) \wedge \exists p'(\mathsf{suggests}(p', p) \wedge \mathsf{in}(p')) \\ 0, & \text{otherwise} \end{cases} \tag{5.33}$$

We can now search for configurations that optimize any of the weight functions or form a linear combination of the values:

$$w(c) = c_p w_p(c) + c_r w_r(c) + c_s w_s(c) \tag{5.34}$$

where c_p, c_r, and c_s are constants that determine the relative importance of the components.

5.8 RECONFIGURATION

The Debian reconfiguration problem has two aspects. First, if there is a new version of the Debian system available, the configuration model must be updated accordingly. Second, we must find a way to change the configuration into one satisfying the new user requirements.

We can change the configuration model quite easily since we do not have to change the inference rules for justifications and constraints. It is enough to add facts that represent the new packages and their relationships and to remove the facts for packages that are removed from the model. However, we may have to remember the facts of more than one Debian version at a time if we for some reason have to use older versions of packages.

By defining a suitable set of reconfiguration operators we can express the other part of a reconfiguration process as a planning problem. In a planning problem we have a set of *states* and a set of *operators* that define the legal ways to change the system from one state into another one, and an *initial state*. The problem is to find a sequence of operators that leads from the initial state to a *goal state*.

We form the state space of the reconfiguration model by taking all $2^{|S|}$ subsets of the set S of packages as states. The initial state corresponds to the packages that are currently installed and the goal state corresponds to a configuration that satisfies the user requirements. We need two operators, one for installing and one for removing packages. The operator $install(p)$ adds the package p to the configuration. We may only install a package if its predependencies are satisfied. The operator $remove(p)$ removes the package p from the configuration.

5.9 CONDITIONAL RELATIONS

In this section we examine the conditional relations. In the beginning we leave the context of the rule language RL for a while and examine the situation at a higher level.

Currently Debian supports two constructs that may be interpreted as conditional relations:

1. alternative recommendations, suggestions, and dependencies; and
2. relations that are restricted by a version clause.

When a package P recommends two alternative packages, A and B, the author of the package thinks that one of them should be installed if P is installed. That is, if A is not installed, the author recommends that B should be installed and vice versa. Thus, we can conceptualize the alternative recommendations as follows:

P recommends A if B is not in the configuration
P recommends B if A is not in the configuration

A version clause asserts that the relation A $\circ$ B between packages A and B holds if the version number of B satisfies a condition. We can remove the conditions from conflict, and providing relations by stating that the relation holds for all packages with compatible versions. For the rest of the relations, a version clause can be seen as defining a set of alternative versions: a package A depends on one particular version of a package B if there is no other compatible version of B in the configuration.

Example 5.9.1 Suppose that there is a relation definition

P depends on A version 1.0 or later

in the configuration model and there are two concrete versions of A available: 1.5 and 2.0. Then we can remove the version clause by defining the following relations:

P depends on A-1.5 if A-2.0 is not in the configuration
P depends on A-2.0 if A-1.5 is not in the configuration

◊

5.9.1 Propositional Language $\mathcal{P}$

In this section we define a propositional language $\mathcal{P}$ which can be used to express conditional relations. We will use the set of software, virtual, and aggregated packages as atomic propositions and complex propositions may be constructed using the normal logical connectives.

We need also a way of speaking about different versions of packages. Here we adopt the syntax used in Debian package description files. A version dependency is indicated by placing a version clause of the form ($op\ version$) after a package name. Here op may be any of the common relational operators $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$. We will adopt the convention that we model concrete packages by using the corresponding software package and a version clause with equality. For example, package gawk-3.1.10 could be represented as proposition gawk $(= 3.1.10)$. We augment the current usage of version clauses by allowing versioning of the virtual packages. A package may provide all or only one version of a virtual package.

Now we have a dilemma in that the language should be expressive enough to cover all interesting use-cases but simple enough that the truth of a proposition can be decided as easily as possible. In particular, we want to ensure that checking whether a configuration is valid stays in P. The truth value of a normal Boolean formula can be checked in polynomial time [28, pp. 81–82] and we would like to have the same property in $\mathcal{P}$. The only construct that causes problems is the version clause; each clause introduces an implicit existential quantification to the formula. Because the set of Debian packages is finite, we can handle the quantification by translating it into a disjunction over all compatible concrete packages. However, in worst case each transformation may increase the size of the formula by n terms, where n is the number of packages in the model. Thus, if we start with a formula with l terms, the size of the formula will grow $O(l \cdot n)$ which is not polynomial anymore. We can ensure that the length stays polynomial by setting a upper limit to the number of connectives in the formula to n which results in $O(n^2)$ growth.

In practice the limit is not important since most of the conditions will be short, involving only two or three packages and there are only few different versions for each software package available at a time.

Definition 5.9.1 *The propositions of language $\mathcal{P}$ are formed in following way:*

1. *All symbols o where o is a software or virtual package are atomic propositions.*

2. *Symbols $\top$ (top) and $\bot$ (bottom) are atomic propositions.*

3. *All symbols of the form $o(op\ v)$, where o is an atomic proposition representing, $op \in \{<, >, \leq, \geq, =, \neq\}$, and v is a version number are propositions.*

4. *If α and β are propositions, then $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, and $(\neg\alpha)$ are propositions.*

5. *A string of symbols is a proposition if and only if it can be constructed by starting with propositions in (1) and (2) and repeatedly applying rules (3–4) at most $|S|$ times, where S is the set of packages in the model.*

We cannot define the semantics for $\mathcal{P}$ in isolation from the version space of the configuration model as otherwise the semantics of version clauses is not well-defined. There are many possible formulations. For example, we can construct version graphs or we can replace the clauses with corresponding disjunction. In the definition below we assume that there exists a way of linking software packages to the concrete packages and a way to extract the version information of a package.

Definition 5.9.2 *A proposition P is* satisfied *in a configuration c (denoted $c \models P$) if and only if one of the following conditions holds:*

1. *for all configurations c, $c \models \top$ and $c \not\models \bot$,*
2. *P is an atomic proposition and $P \in c$,*
3. *P is of the form $o(op\ version)$ and there exists a package $o' \in c$ such that o' is a version of o and its version satisfies the relation,*
4. *P is of the form $(p \wedge q)$ and $c \models p$ and $c \models p$,*
5. *P is of the form $(p \vee q)$ and $c \models p$ or $c \models p$, or*
6. *P is of the form $(\neg p)$ and $c \not\models p$.*

The problem of checking whether $c \models P$ is called $\mathcal{P}$-VALUE.

In Section 5.10 we will show that $\mathcal{P}$-VALUE is, indeed, in P.

5.9.2 Translation of $\mathcal{P}$ to RL

We can express the propositions of $\mathcal{P}$ in RL programs by defining a new predicate p and generating a new instance of it for each proposition.

1. For propositions of the form p where p is a package pck we generate rule

$$\mathsf{p}(\mathsf{x}) \leftarrow \mathsf{in}(\mathsf{pck}) \tag{5.35}$$

   where x is a new constant.

2. For propositions of the form $\mathsf{pck}\ (\geq \mathsf{version})$ we add a rule

$$\mathsf{p}(\mathsf{x}) \leftarrow \exists \mathsf{pck}'(\mathsf{in}(\mathsf{pck}') \wedge \mathsf{available\_version}(\mathsf{pck}, \mathsf{pck}') \wedge \mathsf{pck}' \geq \mathsf{version}) \tag{5.36}$$

   where the relational comparison examines the version string of pck'. We need corresponding rules for each relational operator.

3. For proposition p_1 and p_2 we add a rule

$$\mathsf{p}(\mathsf{x}) \leftarrow \mathsf{p}(1), \mathsf{p}(2). \tag{5.37}$$

where $\mathsf{p}(1)$ and $\mathsf{p}(2)$ correspond to the propositions p_1 and p_2.

4. For proposition p_1 or p_2 we add rules

$$\begin{aligned} \mathsf{p}(\mathsf{x}) &\leftarrow \mathsf{p}(1) \\ \mathsf{p}(\mathsf{x}) &\leftarrow \mathsf{p}(2) \end{aligned} \tag{5.38}$$

5. For propositions of the form not p_1 we add a rule

$$\mathsf{p}(\mathsf{x}) \leftarrow \text{not}\, \mathsf{p}(1) \tag{5.39}$$

Example 5.9.2 Let $S = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$, where $a_1, \ldots, a_n$, $n > 3$ are distinct versions of software package a and $b_1, \ldots, b_m$, $m > 2$ be versions of software package b. Let a proposition P be

$$P = a(>\ 3) \wedge \neg b(\leq 2). \tag{5.40}$$

The translation of P to RL is constructed from inside out as follows

$$\begin{aligned} p(1) &\leftarrow \exists a'(\mathsf{in}(a') \wedge \mathsf{available\_version}(a, a') \wedge a' > 3) \\ p(2) &\leftarrow \exists b'(\mathsf{in}(b') \wedge \mathsf{available\_version}(b, b') \wedge b' \leq 2) \\ p(3) &\leftarrow \text{not}\, p(2) \\ p(4) &\leftarrow p(1), p(3) \end{aligned} \tag{5.41}$$

Now the predicate $p(4)$ is a translation of the proposition P. $\Diamond$

5.10 COMPUTATIONAL COMPLEXITY

In this section we examine the computational complexity of the model presented above. We will examine four problems: VALID-CONFIGURATION, EXISTS-CONFIGURATION, QUERY-CONFIGURATION, and QUERY-RECONFIGURATION.

In VALID-CONFIGURATION we are given a configuration, a configuration model, and a set of user requirements. We are then asked whether the configuration is valid or not.

In EXISTS-CONFIGURATION we are given a configuration model and we have to decide whether it has any valid configurations.

In QUERY-CONFIGURATION we are given a configuration model and a set of user requirements and we want to check whether there exists a configuration that satisfies the requirements. The first of the problems is shown to be in P and the two others are NP-complete.

We begin with showing that the truth value of a $\mathcal{P}$ proposition P can be checked in polynomial time. To prove it, we will show how to reduce the problem to the problem of CIRCUIT VALUE in polynomial time. In the CIRCUIT VALUE problem we are given a directed acyclic graph composed of and-, or-, not-, and input gates. A truth value is then assigned to each leaf node and we have to calculate the truth value of the root node. In [28, pp. 81–82] CIRCUIT VALUE is proven to have a polynomial time algorithm. In fact, it is proven to be P-complete.

Theorem 5.10.1 *There exists a polynomial time reduction $f(c, P)$ from $\mathcal{P}$-VALUE to CIRCUIT VALUE*

$$\textit{such that } c \models P \textit{ if and only if } \text{CIRCUIT VALUE}(f(c, P)) = \texttt{true} \ .$$

Proof. The function $f(c, P)$ is defined recursively as follows:

1. If P is either $\top$ or an atomic proposition o and $o \in c$, then $f(c, P)$ is an input node that is binded to the value true.

2. If P is either $\bot$ or an atomic proposition o and $o \notin c$, then $f(c, P)$ is an input node that is binded to the value false.

3. If P is of the form $(p \wedge q)$, then $f(c, P)$ is an and-gate r and there are edges $(f(c, p), r)$ and $(f(c, q), r)$ in the graph.

4. If P is of the form $(p \vee q)$, then $f(c, P)$ is an or-gate r and there are edges $(f(c, p), r)$ and $(f(c, q), r)$ in the graph.

5. If P is of the form $(\neg p)$, then $f(c, P)$ is a not-gate r and there is the edge $(f(c, p), r)$ in the graph.

6. If P is of the form $o(op\ version)$, then $f(c, P)$ is an or-node r and we add an edge $(f(c, o'), r)$ for each concrete version o' of o that satisfies the version clause.

The construction of the new gate in steps (1) – (5) takes a constant amount of time. We have to execute exactly one of the steps (3) – (5) for each connective in the proposition, so they will be executed $O(l)$ times, where l is the length of the proposition.

In the step (6) we have to potentially examine each package p to determine whether p is a compatible version of o. Thus, this step is $O(n)$. If we have to execute (6) for each leaf node of the graph, the algorithm takes $O(l \cdot n)$ steps, where $l = O(n)$ by definition, giving the total worst case complexity of $O(n^2)$. □

Corollary 5.10.1 *The problem $\mathcal{P}$-VALUE is in* P.

Example 5.10.1 Let a proposition P be defined as in Example 5.9.2. Now the corresponding Boolean circuit is shown in Figure 5.1. ◊

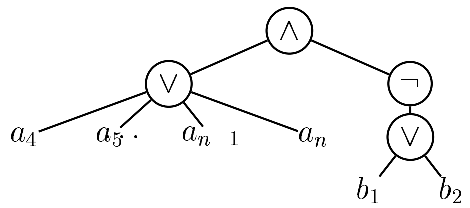


Figure 5.1: A Boolean circuit corresponding to the proposition $a(> \ 3) \wedge \neg b(\leq 2)$.

5.10.2 Configuration Tasks

We saw in Section 3.4 that finding a stable model for a RL program is a NP-complete problem. As in the problems EXISTS-CONFIGURATION and QUERY-CONFIGURATION we want to find a model for a program, they are clearly also NP-complete.

Theorem 5.10.2 *The problem* EXISTS-CONFIGURATION *is* NP-*complete.*

Theorem 5.10.3 *The problem* QUERY-CONFIGURATION *is* NP-*complete.*

Proof(of Theorems 5.10.2 and 5.10.3). The theorems follow directly from Theorem 3.4.1. □

The complexity of problem VALID-CONFIGURATION is not as clear. The examined configuration fixes the atoms that represent the actual configuration but, unfortunately, we cannot use Corollary 3.4.1 directly because we do not know which packages are justified in the configuration. To prove that the problem is in P we have to show that we can handle the justifications in polynomial time.

Theorem 5.10.4 *The problem* VALID-CONFIGURATION *is in* P.

Proof. When we examine the validity of a configuration, we have to check that all packages in it are justified and that all constraints of the model are satisfied.

As the constraints do not depend on justifications of packages, we can check their truth value in polynomial time by Corollary 3.4.1. By Corollary 5.10.1 we know that we can decide which conditional relations hold in the configuration in polynomial time.

As all rules for package justifications are propositional Horn rules, their closure can be calculated in polynomial time [7] and the proof is completed. □

Chapter 6

The Diagnostic Model

In this section we will examine the case where the user requirements cannot be satisfied for some reason. In these cases it is not enough to decide that a suitable configuration does not exist but we need to point out the problem to the user. We will do this by constructing a diagnostic model of the configuration model.

As it is difficult to augment the rules we defined in Chapter 5 in such way that we get both valid configurations and diagnostic messages from the same program, we will divide the Debian configuration process into three stages:

1. First we try to find a configuration that satisfies the user requirements. At this stage we consider only static constraints and invariants.

2. If such configuration is found, we try to find the correct installation order satisfying all dynamic constraints.

3. If there is no satisfying configuration, we construct a diagnosis that explains why the requirements cannot be satisfied.

This process is illustrated in Figure 6.1. In this work we examine only the first and the last step in detail.

6.1 BASIC APPROACH

The usual approach for diagnosis would be to remove problematic components one at a time until a valid model can be found. However, here we go to the other direction. We add a set of new atoms to the program to represent

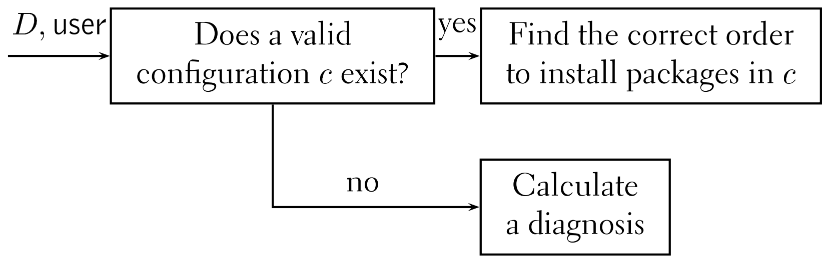


Figure 6.1: The Debian configuration process.

the error conditions and modify the constraints in such way that it will always have at least one model. The new atoms that are true in the model will then identify a set of errors in the requirements.

The new program may have more than one model if there are many possible ways to break the constraints. Each model corresponds to one set of choices that leads to a contradiction. When diagnosing a problem, it is often enough to examine one model, preferably the one that has the least number of problems. The reason for this is that if all our possible choices lead to a contradiction, it does not matter much what particular choices we made. There is only one major requirement. We have to prefer conflicts over missing dependencies, that is, we have to add each packet that is depended on to the configuration even if it causes a conflict. Otherwise we would only notice that some packages are missing but we would not know why they were not included and thus we would miss the conflict.

Example 6.1.1 Consider the situation where the user wants to have packages A and B in a configuration. The package A needs to have either C or D and both C and D conflict with B.

There are three ways to invalidate the constraints. If we add C in the configuration, there is a conflict with B. The same thing happens if we add D. If we leave both out, the dependency relation is not satisfied.

The first two cases give us more information than the third one. In both of them we notice the problem is that adding both A and B leads to a conflict. In the third case we notice that A does not have its dependencies satisfied, but we cannot directly see what caused the problem. ◊

We need predicates to express dependency and conflict constraints. In addition, we would like to have a set of predicates that tells for any given package why it is added to a configuration. This is useful because there might be two long chains of dependencies that finally lead to a conflict. Without explicit extra information about the dependencies the user may have a difficult time trying to find the real cause of the problem. However, we need also a mechanism to suppress the dependency information if it is not required since there would easily be hundreds or thousands of explanations otherwise.

6.2 DIAGNOSTICS

We will use a number of new atoms to represent the diagnosis of a program. The atoms can be divided into two classes: *problems* and *explanations*. The problems explain what is wrong with the user requirements and the an explanations tell us why a package was taken into the model.

There are two possible problems:

1. missing(P): The package P should be in a configuration, but for some reason it could not be added to it.

2. in_conflict(P_1, P_2): The packages P_1 and P_2 conflict with each other and they both had to be taken into the configuration.

There are three possible explanations:

1. $\mathsf{user\_selected}(P)$: The user explicitly selected P to be in the configuration.
2. $\mathsf{is\_required}(P)$: The package P is of *required* priority.
3. $\mathsf{needs}(P_1, P_2)$: The package P_2 is in the configuration because P_1 depends on it.

6.3 MISSING PACKAGES

We will use the predicate missing to denote unsatisfied dependencies. As we take the approach that we add all necessary packages to the configuration model, a package may be missing only if it is not present in the configuration model at all. We will treat all packages without definitions as virtual packages and consequently only virtual packages or some specific versions of concrete packages may be missing.

We have to replace the dependency constraint from the configuration model with two rules.

1. If a package P_1 depends on a package P_2, P_2 has to be in a configuration when P_1 is in it.

$$\mathsf{in}(P_2) \leftarrow \mathsf{in}(P_1), \mathsf{depends}(P_1, P_2) \tag{6.1}$$

2. If we have to add a virtual package V with no providers, that package is missing.

$$\mathsf{missing}(V) \leftarrow \mathsf{in}(V), \mathsf{virtual}(V), \neg\exists P(\mathsf{provides}(P, V)) \tag{6.2}$$

6.4 CONFLICTS

We will use the predicate in_conflict to model conflicting requirements. The atom $\mathsf{in\_conflict}(P_1, P_2)$ is true exactly when P_1 and P_2 conflict with each other and they both have to be in the configuration.

$$\mathsf{in\_conflict}(P_1, P_2) \leftarrow \mathsf{conflicts}(P_1, P_2), \mathsf{in}(P_1), \mathsf{in}(P_2) \tag{6.3}$$

We must also handle the situation where the user explicitly wants to leave out a package that some other package needs. We will use a special constant user_exclude and the rule

$$\mathsf{in\_conflict}(P, \mathsf{user\_exclude}) \leftarrow \mathsf{in}(P), \mathsf{user\_exclude}(P) \tag{6.4}$$

to model these cases.

6.5 JUSTIFICATIONS

When a package is missing from a configuration or two packages are in conflict, we would like to know why the problematic packages are necessary. In the base case a package is necessary if it is a required one or if the user explicitly chose it to be in the configuration. A package may also be required if it some mandatory package depends on it.

We will use the predicate needs_reason to mark out the atoms that we want to justify. We can find the explanations by first marking the packages that directly cause problems and then marking all packages that depend on marked packages and are present in the configuration.

We can find the packages that need a reason with the following rules:

1. A missing package needs a reason.

$$\mathsf{needs\_reason}(\mathsf{P}) \leftarrow \mathsf{missing}(\mathsf{P}) \tag{6.5}$$

2. A package in conflict needs a reason.

$$\begin{aligned}\mathsf{needs\_reason}(\mathsf{P}_1) &\leftarrow \mathsf{in\_conflict}(\mathsf{P}_1, \mathsf{P}_2)\\ \mathsf{needs\_reason}(\mathsf{P}_2) &\leftarrow \mathsf{in\_conflict}(\mathsf{P}_1, \mathsf{P}_2)\end{aligned} \tag{6.6}$$

3. A package P_1 needs a reason if it depends on a package P_2 that needs a reason.

$$\mathsf{needs\_reason}(\mathsf{P}_1) \leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{needs\_reason}(\mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{6.7}$$

We will model package justifications with three predicates: user_selected, is_required, and needs. The corresponding rules are:

1. A package is necessary, if the user selected it explicitly.

$$\mathsf{user\_selected}(\mathsf{P}) \leftarrow \mathsf{needs\_reason}(\mathsf{P}), \mathsf{user\_include}(\mathsf{P}) \tag{6.8}$$

2. A package is necessary, if it is a required package.

$$\mathsf{is\_required}(\mathsf{P}) \leftarrow \mathsf{needs\_reason}(\mathsf{P}), \mathsf{required}(\mathsf{P}) \tag{6.9}$$

3. A package P_2 is necessary, if some necessary package P_1 depends on it.

$$\mathsf{needs}(\mathsf{P}_1, \mathsf{P}_2) \leftarrow \mathsf{needs\_reason}(\mathsf{P}_2), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \tag{6.10}$$

6.6 COMPUTATIONAL COMPLEXITY

In the problem DIAGNOSE-CONFIGURATION we want to find a diagnostic model for a Debian configuration model and a set of user requirements. Because the diagnostic model is constructed with RL rules, the problem is NP-complete.

Theorem 6.6.1 *The problem* DIAGNOSE-CONFIGURATION *is* NP*-complete.*

Proof. The theorem follows directly from Theorem 3.4.1. □

Chapter 7

Detailing the Model

In this section we flesh out the implementation details of the Debian configuration model using the rule language RL. We start by describing some changes that we have to make to allow the model to work efficiently. Then we describe the main points of the translation in detail. However, we do not present the whole translation here since there are many almost identical rules. The corresponding `smodels` program can be found in Appendix A. However, the program does not implement the rules for aggregate packages.

7.1 CHANGES TO THE MODEL

The main computational problem for the Debian configuration model we defined in Chapter 5 comes from the optimization functions. The implementation of `smodels` handles weight maximization by generating a series of models where each successive model has higher weight than its predecessors. As this potentially makes the computation task much more difficult, we circumvent it by adding explicit rules for the preference handling.

We can fix the situation by adding rules that state that an important or a standard package is included in the configuration if nothing else prevents it.

$$\mathsf{in}(\mathsf{P}) \leftarrow \mathsf{important}(\mathsf{P}), \mathsf{can\_have}(\mathsf{P}) \tag{7.1}$$

We will take also the same approach for the recommendations and suggestions. We will add a recommended package to a configuration if it does not cause any problems.

$$\mathsf{in}(\mathsf{P}_2) \leftarrow \mathsf{recommends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{can\_have}(\mathsf{P}_1) \tag{7.2}$$

We add two new atoms, deny_recommendations and allow_suggestions, to control whether the recommendations and suggestions are used or not.

We need to define a complementary predicate to can_have(P) that defines when we cannot add P to the configuration. For this, we will use the predicate cant_have(P). The reason for including both can_have and cant_have may not be intuitively clear. However, the stable model semantics has the property of implicit negation: if we cannot prove an atom we can infer that its negation is true. This causes problems if a package recommends a package name that is not available at all. In that case we would erroneously conclude that we could add the recommended package since not cant_have(name) would be trivially true. We define the rule

$$\mathsf{can\_have}(\mathsf{P}) \leftarrow \mathsf{package}(\mathsf{P}), \mathsf{not\ cant\_have}(\mathsf{P}) \tag{7.3}$$

for can_have.

A short summary of all predicates that are used in finding valid configurations is presented in Table 7.1. Table 7.2 gives a summary of additional predicates that are used with the diagnostic model.

7.2 PACKAGES

Most methods that we defined in Section 5.2 for modeling packages can also be used in practice without modifications. The concrete packages form the only exception since we have to choose how we want to represent the versions.

We will take the approach that the concrete packages are modeled with a pair of arguments. The first argument denotes the corresponding software package and the second argument identifies the version string. Thus, if we want to add a concrete package package-version to the model, we do it by adding the following two facts to the program.

$$\begin{aligned} \mathsf{software}(\mathsf{package}) &\leftarrow \\ \mathsf{available\_version}(\mathsf{package}, \mathsf{version}) &\leftarrow \end{aligned} \tag{7.4}$$

7.3 RELATIONS

We will model all relations using two types of predicates:

1. domain predicates that tell what relations exist, and
2. controlling predicates that tell when the relations hold.

We need both types since our front-end, `lparse`, only works with domain-restricted programs. That is, each variable that occurs on a rule has to occur also in a positive non-recursively defined literal. Thus, a domain predicate in the rule body gives us those relations that may hold and the controlling predicates tell us what relations are enabled. If the latter type of predicates were left out, we could not express conditional relations.

For example, the relation $\mathsf{depends}(\mathsf{C}_1, \mathsf{P}_2)$ where C_1 is a concrete package and P_2 is a software, virtual, or aggregated package will be encoded using two predicates:

$$\begin{aligned} &\mathsf{depends}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2) \\ &\mathsf{enabled\_depend}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2) \end{aligned} \tag{7.5}$$

If the relation is not conditional, we add both atoms as facts to the program. Otherwise, we add a rule of the form

$$\mathsf{enabled\_depend}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2) \leftarrow \mathsf{p}(\mathsf{x}) \tag{7.6}$$

Predicate	Purpose
aggregate(P)	P is an aggregate package
aggregate_version(P,V)	A version V of an aggregate package P is in the configuration model.
allow_suggestions	If true, include all possible suggested packages into a configuration.
available_version(P,V)	A version V of a package P is in the model.
can_have(P,V)	The package P-V can be added to the configuration without causing problems.
cant_have(P,V)	The package P-V will cause a conflict if it is added to the configuration.
concrete_version(P,V)	There is a concrete package P-V in the model.
conflicts(P1, V1, P2)	The package P1-V1 conflicts with the package P2.
contains(P1, V1, P2)	The aggregate package P1-V1 contains P2.
deny_recommendations	If true, do not include recommended packages in a configuration.
depends(P1, V1, P2)	The package P1-V1 depends on the package P2.
enabled_*relation* (P1, V1, P2, V2)	The relation between package P1-V1 and V2-P2 is enabled in the configuration. Possible relations are: depends, conflicts, recommends, suggests, contains.
has_provider(P)	The virtual package P has a provider in the configuration.
important(P)	P has important priority.
in(P,V)	The package P-V is in the configuration.
object(P)	P is an object in the configuration model.
op_relation(P1,V1,P2,V2)	The package P1-V1 relates to specific versions of P2. The possible operators are $<$, $\leq$, $=$, $\neq$, $\geq$.
p(X)	X is a proposition of $\mathcal{P}$.
package(P)	P is a package.
provides(P1, V1, P2)	The package P1-V1 provides the virtual package P2.
recommends(P1, V1, P2)	The package P2-V1 recommends the package P2.
required(P)	P is a required package
software(P)	P is a software package
standard(P)	P has standard priority
suggests(P1, V1, P2)	The package P1-V1 suggests the package P2.
virtual(P)	P is a virtual package
virtual_version(P, V)	A version V of a virtual package P is in the configuration model.

Table 7.1: List of all predicates used in the translation

Predicate	Purpose
depended_on (P2, V2, P1, V1)	The package P1-V1 may depend on P2-V2.
in_conflict(P1, V1, P2)	The conflicting packages P1-V1 and P2 both must be in a configuration.
is_required(P)	P is in the configuration because it is a required package.
missing(P,V)	The package P-V should be in the configuration but for some reason it is not in it.
needs(P1, V1, P2)	The package P2 is in the configuration because P1-V1 depends on it.
needs_reason(P, V)	The package P-V causes some problem and we need to find a reason why it is included.
possibly_missing(P,V)	Some package depends on P-V and thus it may cause problems if it is not present.
user_selected(P,V)	The package P-V is in the configuration because the user selected it.

Table 7.2: List of predicates that are used in the diagnostic model.

where $\mathsf{p}(\mathsf{x})$ is the translation of the controlling proposition.

The version clauses are modeled by encoding the relational operator into the predicate symbol. For example, we encode the relation

$$\mathsf{A\text{-}1.0} \text{ depends on } \mathsf{B}\ (\geq 1.5)$$

using the predicate

$$\texttt{ge\_depends(A, 1.0, B, 1.5).} \tag{7.7}$$

The possible prefixes for the relation name are lt_, le_, eq_, neq_, ge_, and gt_ and they correspond to the operators $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$, respectively.

7.4 CONSTRAINTS

We will complete the constraints by adding the necessary domain predicates to the rule bodies and detailing the usage of version clauses. As we use similar techniques for each constraint, we will present fully only the rules for handling dependencies and conflicts. We will mention only the interesting points of other constraints.

Dependencies

We define the basic dependency constraint without version clauses with the following rules.

$$\begin{aligned} \leftarrow\, & \mathsf{depends}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2), \\ & \mathsf{enabled\_depend}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2), \\ & \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1), \\ & \mathsf{not\ in}(\mathsf{P}_2) \end{aligned}$$

We can change the above rule to model the version clauses by changing the literal $\mathsf{not\ in}(\mathsf{P}_2)$ into an existential literal that states no compatible versions of P_2 are present in the configuration.

$$
\begin{aligned}
\leftarrow\, & \mathsf{lt\_depends}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\
& \mathsf{enabled\_depend}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\
& \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1), \\
& \neg \exists \mathsf{V}(\mathsf{available\_version}(\mathsf{P}_2, \mathsf{V}) \wedge \mathsf{V} < \mathsf{V}_2 \wedge \mathsf{in}(\mathsf{P}_2, \mathsf{V}))
\end{aligned}
\tag{7.8}
$$

Naturally, we need to duplicate this rule for each relational operator. These rules can be translated into `smodels` rules of the form

$$
\begin{aligned}
\leftarrow\, & 0 \leq \{\mathsf{in}(\mathsf{P2}, \mathsf{V}) : \mathsf{available\_version}(\mathsf{P2}, \mathsf{V}) : \mathsf{vc}(\mathsf{lt}, \mathsf{V}, \mathsf{V2})\} \leq 0, \\
& \mathsf{lt\_depends}(\mathsf{P1}, \mathsf{V1}, \mathsf{P2}, \mathsf{V2}), \\
& \mathsf{enabled\_depend}(\mathsf{P1}, \mathsf{V1}, \mathsf{P2}, \mathsf{V2}), \\
& \mathsf{in}(\mathsf{P1}, \mathsf{V1})
\end{aligned}
\tag{7.9}
$$

Here vc is an external function that implements the version comparison algorithm of the Debian system.

Conflicts

The rules for conflicts are otherwise similar than the ones for dependencies, but we have to add the rules to tell that we cannot add a package if it would conflict with another package that is already in the configuration.

The basic rule for conflicts is almost unmodified.

$$
\begin{aligned}
\leftarrow\, & \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1), \\
& \mathsf{in}(\mathsf{P}_2), \\
& \mathsf{conflicts}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2), \\
& \mathsf{enabled\_conflict}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2)
\end{aligned}
\tag{7.10}
$$

We add the version clauses using rules of the form

$$
\begin{aligned}
\leftarrow\, & \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1), \\
& \mathsf{lt\_conflicts}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\
& \mathsf{enabled\_conflict}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\
& \mathsf{available\_version}(\mathsf{P}_2, \mathsf{V}), \\
& \mathsf{V} < \mathsf{V}_2, \\
& \mathsf{in}(\mathsf{P}_2, \mathsf{V})
\end{aligned}
\tag{7.11}
$$

If we have a package P_1-V_1 in a configuration, we cannot add any conflicting packages to the configuration.

$$
\begin{aligned}
\mathsf{cant\_have}(\mathsf{P}_2) \leftarrow\, & \mathsf{conflicts}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2), \\
& \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1), \\
& \mathsf{enabled\_conflict}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2)
\end{aligned}
\tag{7.12}
$$

Virtual packages
The virtual packages present a new problem for implementation. The existential literal in the constraint

$$\leftarrow \mathsf{in}(\mathsf{V}), \mathsf{virtual}(\mathsf{V}), \neg\exists \mathsf{C}(\mathsf{provides}(\mathsf{C}, \mathsf{V}) \wedge \mathsf{in}(\mathsf{C})) \tag{7.13}$$

contains two non-domain literals and so it cannot be expressed with `lparse` conditional literals. To handle the situation correctly we add a new predicate

$$\texttt{has\_provider(V)} \tag{7.14}$$

to the model. An atom `has_provider(V)` is true in a model if the virtual package V has a provider in the configuration.

We can handle the situation by adding the following two rules to the model

$$\leftarrow \mathsf{virtual}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_1), \text{not } \mathsf{has\_provider}(\mathsf{P}_1) \tag{7.15}$$

$$\begin{aligned} \mathsf{has\_provider}(\mathsf{P}_2) \leftarrow\, & \mathsf{provides}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2), \\ & \mathsf{enabled\_provide}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2), \\ & \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1) \end{aligned} \tag{7.16}$$

7.5 THE DIAGNOSTIC MODEL

The implementation of the diagnostic model follows the basic ideas that were presented above. The only tricky part is the handling of missing packages.

We can handle the cases where a software or virtual package is missing easily, but constructing the rules for a situation where some specific concrete version is missing is more difficult.

First, we have to express that a concrete package is missing if there is no compatible version in the configuration.

$$\begin{aligned} \mathsf{missing}(\mathsf{P}_2, \mathsf{V}_2) \leftarrow\, & \neg\exists \mathsf{V}(\mathsf{in}(\mathsf{P}_2, \mathsf{V}) \wedge \mathsf{available\_version}(\mathsf{P}_2, \mathsf{V}) \wedge \mathsf{V} < \mathsf{V2}), \\ & \mathsf{lt\_depends}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\ & \mathsf{enabled\_depend}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\ & \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1) \end{aligned} \tag{7.17}$$

Unfortunately, this rule allows us to leave a necessary package out even if it is available, since `smodels` could decide to leave a justified package out in favor of an atom `missing(P,V)`. We have to add a rule to forbid this from happening

$$\begin{aligned} \leftarrow\, & \exists \mathsf{V}(\mathsf{in}(\mathsf{P}_2, \mathsf{V}) \wedge \mathsf{available\_version}(\mathsf{P}_2, \mathsf{V}) \wedge \mathsf{V} < \mathsf{V2}), \\ & \mathsf{missing}(\mathsf{P}_2, \mathsf{V}_2) && (7.18) \\ & \mathsf{lt\_depends}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), && (7.19) \\ & \mathsf{enabled\_depend}(\mathsf{P}_1, \mathsf{V}_1, \mathsf{P}_2, \mathsf{V}_2), \\ & \mathsf{in}(\mathsf{P}_1, \mathsf{V}_1) \end{aligned}$$

7.6 ANALYSIS

The model above catches the most important aspects of the Debian configuration management. The number of distinct rules is relatively small even in the diagnostic model. The number of knowledge base items is big but the translation can be fully automated: the information for all packages is stored in a single structured file that can be read and processed by a simple compiler. The core rules are the same for each Debian version and only the knowledge base has to be constructed.

The translation takes a simplified approach to the optimization functions as we can choose only whether we want to include the recommended or suggested packages or not. The smodels implementation supports a simple form of optimization where we can define a set of atoms and maximize or minimize the number of atoms from the set that are in a model. Since we would then have to iterate through all valid configuration of the configuration model, the amount of work would increase significantly.

There are a few problems still left in the model. The most notable ones are:

1. It is possible that an old version of a package is chosen to be installed even if a new one is available if no packages specifically depend on the new version. Currently, `dselect` circumvents this problem by allowing to select only the newest version of a package. Older versions have to be installed by calling `dpkg` by hand.

2. If one of two conflicting packages is recommended and the other is suggested in a configuration, we would like to include the recommended one, but currently either one could be chosen.

The first case could probably be circumvented by adding a new optimization function that prefers newer versions of packages and then using the `smodels` maximize statement.

The final weakness of the translation is a more general problem of the Debian configuration management. The process is very vulnerable to errors in the configuration database. As there are more than 300 maintainers each working on a relatively small number of packages, it is very possible that the description file contains errors and omissions. As a practical example, according to the database conflict is not necessary a symmetric relation if one of the maintainers is not aware of the conflict.

7.7 IMPLEMENTING THE TRANSLATION

The Debian configuration process is implemented with three steps:

1. *preprocessing*,

2. *model generation*, and

3. *postprocessing*.

In the first step we read the Debian configuration file and the user requirements to generate a logic program. After that we run the program through `lparse` and `smodels` to find out whether the system has any stable models. If such model is found, we extract the relevant information from the model using a postprocessing script. Otherwise, we have to construct the diagnostic program with the same steps.

7.7.1 Preprocessing

The preprocessing stage is done in two steps. First we filter out inconsistencies from the package description file using `change_list_format`-script. Then we generate the logic program of the model using a simple compiler, `translator`.

As we saw in Example 4.4.3 there are many internal inconsistencies in the configuration file which are a result of the behavioral semantics of `dpkg`. We remove the inconsistencies by filtering out all relation definitions where a package conflicts with itself or with a package it provides. At the same time we remove all unallowed symbols from the package names and replace them with legal ones.

The `translator` reads in the filtered package description file and generates the knowledge base that is used for finding valid configurations. The `translator` is written in C++ using the Standard Template Library [42] and the Bison parser generator. The `change_list_format` is written in Perl.

7.7.2 Model Generation

In the next step we generate the actual configuration by grounding the logic program and the user requirements using `lparse` and searching a stable model with `smodels`.

The version comparison algorithm `vc` is taken directly from the `dpkg` source files. The `lparse` allows the user to define arbitrary numerical functions to be called from the logic program. Unfortunately, this is not enough for `vc` since it needs to compare two text strings. Thus, we have to patch `lparse` source code directly to add the function there.

7.7.3 Postprocessing

As the stable model contains many more atoms than those representing the configuration, we extract the configuration from the model using a Perl-script `extract_configuration`. The script scans through the stable models and removes all unwanted atoms. The postprocessing stage is included only for convenience and it is only a simple filter.

7.7.4 Example Run

Here we consider a simple example of how the implementation works. In the configuration model we have three packages, t_1, t_2, and t_3. Here t_1 is a required package which depends on an optional package t_2 and a virtual package v_1 which is provided by t_3. Below we see the situation after the package description file has gone through `extract_configuration` but before `translator`.

```
package t_1 {
 version: 1.0
 priority: required
 depends: t_2 (>= 1.1), v_1
}
package t_2 {
 version: 1.1
 priority: optional
}
package t_3 {
 version: 1.2
 priority: optional
 provides: v_1
}
```

The configuration model is then processed by `translator`, `lparse`, `smodels`, and finally by `extract_configuration`:

```
filter % ./translator < package_data > cnf_model
filter % lparse cnf_model | smodels > configuration
filter % ./extract_configuration < configuration
Model 1
in(t_1)
in(t_3)
in(t_3,"1.2")
in(t_2)
in(t_2,"1.1")
in(t_1,"1.0")
in(v_1)
```

As we see, the concrete packages t_1-1.0, t_2-1.1, and t_3-1.2 are in the configuration with the virtual package v_1. The line `Model 1` tells us that the configuration was the first one that was found. It is possible to use command line arguments to tell `smodels` to find all valid configurations.

Chapter 8

Evaluation

In this section we evaluate the configuration model we generated in the previous section by running it using the actual data from the Debian GNU/Linux versions 2.0 and 2.1.

8.1 TEST CASES

The implementation was tested using the package data description of Debian versions 2.0 and 2.1. Only the packages from the main distribution were used with the version 2.0 containing 1526 and 2.1 containing 2260 packages.

The user requirements were simulated by randomly selecting a number of packages to be included in the configuration. While random choices most probably do not model very well the requirements of a real user, they at least give us some picture of the practicality of the model.

The random choice was made by including a probability parameter p ($p < 1$) for the translation. A uniformly distributed random number r between $[0, 1]$ was generated for each concrete package, and if $r \leq p$, the package was included into user requirements.

If the user requirements were satisfiable, only the valid configuration was computed. Otherwise, the diagnostic output was also generated for the problem. If the reason for unsatisfiability was conflicting requirements, the number of conflicts was also counted.

The size of the grounded programs is shown in Table 8.1.

Problem	Packages	Atoms	Rules
Debian 2.0	1526	25996	28443
Debian 2.0 Diagnostic	1526	29428	31968
Debian 2.1	2260	37958	41286
Debian 2.1 Diagnostic	2260	42915	46377

Table 8.1: The size of the test cases after they have been grounded. The program sizes are for a case where there are no user requirements at all.

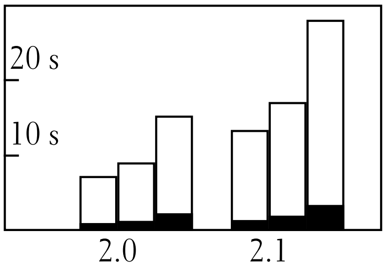


Figure 8.1: Results from the test runs. The bars are from left to right: the average time to find a configuration if such exists, the average time of all test cases, and the average time when a diagnosis is needed. The shaded areas correspond to the time that `smodels` used.

8.2 TEST RESULTS

The tests were run on a 233 MHz Intel Pentium II with 128 MB of main memory running Debian GNU/Linux version 2.1. The stable models were computed using `smodels` version 2.23 and the program was grounded using `lparse` version 0.99.40.

The tests were run using four different values for the parameter p: 0.2, 0.1, 0.02, and 0.01. For each test case 40 sets of user requirements were generated and each individual case was run five times. The execution times were measured with the `time` utility by summing `user` and `system` times.

The results of the runs are shown in Table 8.2. The random test cases are divided into two classes: those that have a valid configuration and those that have not. The number of choice points is left out of the table because it was very small in all cases. There was one choice point for each user-selected package that depended on a virtual package with many providers.

The time division between grounding and model generation steps is presented in graphical format in Figure 8.1. The cases where it was necessary to generate a diagnosis are summarized in Table 8.3.

8.3 ANALYSIS

The results showed that the grounding of the logic program took almost an order of magnitude more time than the actual computation of the stable

Problem	Number	Time (s)	GTime (s)	MTime (s)
Debian 2.0	125	7.33	6.27	1.06
Debian 2.0 Diagnostic	35	15.31	12.95	2.36
Average	160	9.08	7.73	1.35
Debian 2.1	120	13.41	11.95	1.46
Debian 2.1 Diagnostic	40	27.97	24.51	3.46
Average	160	17.05	15.01	2.04

Table 8.2: The test results. GTime tells the time used in grounding the problem and MTime shows how much time `smodels` used in finding models.

Problem	p	Number	% of Total	Conflicts	Conflicts/Cases
Debian 2.0	0.2	17	42.5%	26	1.153
	0.1	13	32.5%	13	1
	0.02	3	7.5%	3	1
	0.01	2	5.0%	2	1
total	–	29	18.1%	31	1.07
Debian 2.1	0.2	23	57.5%	32	1.391
	0.1	12	30.0%	12	1
	0.02	2	5.0%	2	1
	0.01	3	7.5%	3	1
total	–	40	25.0%	49	1.225

Table 8.3: Analysis of the cases where the user requirements were unsatisfiable.

models. While the grounding took more than 10 seconds in each case, the stable models could be computed in a second or two. This presents a problem for the practical use of the model since the 10 second wait would be unacceptable for most users, especially when the current system with all its shortcomings can analyze the configurations much faster.

If the time spent on grounding can be shortened, the model becomes more viable for actual use. As the current version of `lparse` does not offer support for incremental grounding, that is, each set of user requirements must be grounded individually and the old grounding has to be discarded, we have to seek other approaches. It may be possible to ground the program only once with all possible user requirements selected and then filter out those selections that are not made. Currently, `smodels` offers an programming interface that may be utilized in this task. With this approach we would ground the program once each time when the Debian configuration model changes. As this happens quite infrequently, a ten second wait is quite acceptable.

The results also showed that the cases where a diagnosis had to be generated took approximately twice as much time as the simple cases. This seems to suggest that the problems of finding a valid configuration and finding a diagnosis are equally difficult to compute in practice. In most cases there was only one conflict in the model and the user would have had to change one requirement to remove it. While the requirements of a real user probably differ much from the ones generated randomly, the approach seems promising.

Finally, the number of choice points was very low in all cases. Most of the models had no choice points at all and the only choices resulted from cases where an user-selected package depended on a virtual package that had many providers. This is an important property since each individual choice point potentially doubles the execution time. However, it is probable that in practical use there would be more choice points since the user might want to express the requirements using mostly virtual packages.

Chapter 9

Conclusions

In this work we saw a brief glimpse of the current state of the art on the configuration management problem and we introduced a new rule-based language RL for representing configuration knowledge. The rule language is based on stable model semantics and it allows finite existential quantification and choice rules. The decision problem for RL was shown to be NP-complete.

We examined the current status of the Debian GNU/Linux configuration management and proposed a number of improvements on it. As a first improvement we provided a formal semantics for the relation descriptions in the form of a configuration model. We also added two new package types, software and aggregated packages to the model. A software package represents some specific software item that can be provided by a concrete package in the distribution. An aggregated package represents a combination of other packages that provides some large functionality. We allowed the relationships to be defined for virtual and aggregated packages in addition to current practice of defining relations only for concrete packages. As the final improvement, we introduced the notion of conditional relationships. A conditional relation holds only if some external condition is in effect.

We showed that if we are given a Debian GNU/Linux configuration model and a set of user requirements, then the problem of deciding whether the configuration is valid is in P. On the other hand, the process of finding a configuration satisfying the requirements is NP-complete.

We presented a diagnostic model that can be used to find unsatisfiable user requirements. We can use the model to find conflicting and missing packages and we get an explanation why the packages are required to be in the configuration. The diagnostic model is not complete in the sense that it does not necessarily find all problems at once so the user may have to change his requirements a couple of times until a suitable configuration is found.

We evaluated the configuration model with the actual configuration data from Debian versions 2.0 and 2.1. The results showed that `smodels` could find a configuration for 2260 packages in a few seconds. This is a very promising result and it seems that the model could be used also in practice. However, the grounding time of the model was too long and we have to improve it vastly before we can apply the model for real cases. We presented a way to reduce the grounding time by grounding the program only once with all possible user requirements and then filtering the unnecessary requirements out of it.

We showed that the Debian configuration model could be generated automatically from the description files. The logical next step for research would be construction of a practical configuration management tool that utilizes the model. The tool would need two components: a graphical interface to allow the user to make her selections and a back end to perform the actual configuration maintenance. It is possible that some of the existing tools can be modified to use the model.

Appendix A

Full Configuration Model

```
%% Debian GNU/Linux configuration model

allow_suggestions.

%% A package is either a software or virtual package
package(X) :- software(X).
package(X) :- virtual(X).

%% If we have an unidentified package, we treat is as a software
%% package.
virtual(X) :- object(X), not software(X).

%% Both concrete and virtual versions are available.
available_version(X,Y) :- concrete_version(X,Y).
available_version(X,Y) :- virtual_version(X,Y).

%% A package may be in a configuration, if it can be justified.
{ in(X) } :- package(X), justified(X).
{ in(X,Y) } :- available_version(X,Y), justified(X,Y).

%% The dependency constraint

  :- depends(P1, V1, P2),
     in(P1, V1),
     enabled_depend(P1, V1, P2),
     not in(P2).

  :- { in(P2, V) : available_version(P2, V)
                 : vc(lt, V, V2) } 0 ,
     lt_depends(P1, V1, P2, V2),
     enabled_depend(P1, V1, P2, V2),
     in(P1, V1).

  :- { in(P2, V) : available_version(P2, V)
                 : vc(le, V, V2) } 0 ,
     le_depends(P1, V1, P2, V2),
```

```
    enabled_depend(P1, V1, P2, V2),
    in(P1, V1).

  :- { in(P2, V) : available_version(P2, V)
                 : vc(gt, V, V2) } 0 ,
    gt_depends(P1, V1, P2, V2),
    enabled_depend(P1, V1, P2, V2),
    in(P1, V1).

  :- { in(P2, V) : available_version(P2, V)
                 : vc(ge, V, V2) } 0 ,
    ge_depends(P1, V1, P2, V2),
    enabled_depend(P1, V1, P2, V2),
    in(P1, V1).

%% The conflict constraint
  :- conflicts(P1, V1, P2),
    in(P1, V1),
    in(P2),
    enabled_conflict(P1, V1, P2).

  :- 1 { in(P2, V) : available_version(P2, V)
                   : vc(lt, V, V2) },
    lt_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).

  :- 1 { in(P2, V) : available_version(P2, V)
                   : vc(le, V, V2) },
    le_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).

  :- 1 { in(P2, V) : available_version(P2, V)
                   : vc(gt, V, V2) },
    gt_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).

  :- 1 { in(P2, V) : available_version(P2, V)
                   : vc(ge, V, V2) },
    ge_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).


% Virtual package handling
  :- virtual(P1),
    not has_provider(P1),
```

```
     in(P1).

   :- provides(P1, V1, P2),
      enabled_provide(P1, V1, P2),
      in(P1,V1),
      not in(P2).

has_provider(P2) :-
    provides(P1, V1, P2),
    enabled_provide(P1, V1, P2),
    in(P1,V1).

%% The concrete package constraint
 :- { in(P, V) : available_version(P, V) } 0,
    software(P),
    in(P).

 :- not in(P),
    concrete_version(P, V),
    in(P,V).

 :- 1 { in(P,V2) : available_version(P, V2)
                 : V1 != V2 },
    concrete_version(P, V1),
    in(P,V1).

%% The required package constraint
 :- required(P), not in(P).

%% User requirements
  :- user_include(P), not in(P).
  :- user_include(P,V), not in(P,V).
  :- user_exclude(P), in(P).
  :- user_exclude(P,V), in(P,V).

%% Justification rules
justified(P) :- standard(P).
justified(P) :- optional(P).
justified(P) :- required(P).
justified(P) :- user_include(P).
justified(P,V) :- user_include(P,V).

% A package is justified if some other package depends on it
justified(P2) :-
    depends(P1, V1, P2),
    enabled_depend(P1, V1, P2),
    in(P1, V1).

justified(P2, V) :-
```

```
    gt_depends(P1, V1, P2, V2),
    enabled_depend(P1, V1, P2, V2),
    available_version(P2, V),
    vc(gt, V, V2),
    in(P1, V1).

justified(P2, V) :-
    lt_depends(P1, V1, P2, V2),
    enabled_depend(P1, V1, P2, V2),
    available_version(P2, V),
    vc(lt, V, V2),
    in(P1, V1).

justified(P2, V) :-
    ge_depends(P1, V1, P2, V2),
    enabled_depend(P1, V1, P2, V2),
    available_version(P2, V),
    vc(ge, V, V2),
    in(P1, V1).

justified(P2, V) :-
    le_depends(P1, V1, P2, V2),
    enabled_depend(P1, V1, P2, V2),
    available_version(P2, V),
    vc(le, V, V2),
    in(P1, V1).

% A package is justified if it provides a package that is needed.
justified(P1, V1) :-
    provides(P1, V1, P2),
    enabled_provide(P1, V1, P2),
    in(P2).

% A virtual package is justified if its provider is in the
% configuration.
justified(P2) :-
    provides(P1, V1, P2),
    enabled_provide(P1, V1, P2),
    in(P1, V1).

% Concrete versions of an included software package are justified
justified(P, V) :-
    available_version(P, V),
    in(P).

% and vice versa.
justified(P) :-
    available_version(P, V),
    in(P,V).
```

```
% Recommended and suggested packages are justified.
justified(P2) :-
    recommends(P1, V1, P2),
    enabled_recommendation(P1, V1, P2),
    in(P1, V1).

justified(P2, V) :-
    gt_recommends(P1, V1, P2, V2),
    enabled_recommendation(P1, V1, P2, V2),
    available_version(P2, V),
    vc(gt, V, V2),
    in(P1, V1).

justified(P2, V) :-
    lt_recommends(P1, V1, P2, V2),
    available_version(P2, V),
    enabled_recommendation(P1, V1, P2, V2),
    vc(lt, V, V2),
    in(P1, V1).

justified(P2, V) :-
    ge_recommends(P1, V1, P2, V2),
    available_version(P2, V),
    enabled_recommendation(P1, V1, P2, V2),
    vc(ge, V, V2),
    in(P1, V1).

justified(P2, V) :-
    le_recommends(P1, V1, P2, V2),
    available_version(P2, V),
    enabled_recommendation(P1, V1, P2, V2),
    vc(le, V, V2),
    in(P1, V1).

justified(P2) :-
    suggests(P1, V1, P2),
    enabled_suggestion(P1, V1, P2),
    in(P1, V1).

justified(P2, V) :-
    gt_suggests(P1, V1, P2, V2),
    available_version(P2, V),
    enabled_suggestion(P1, V1, P2, V2),
    vc(gt, V, V2),
    in(P1, V1).

justified(P2, V) :-
    lt_suggests(P1, V1, P2, V2),
```

```
    enabled_suggestion(P1, V1, P2, V2),
    vc(lt, V, V2),
    available_version(P2, V),
    in(P1, V1).

justified(P2, V) :-
    ge_suggests(P1, V1, P2, V2),
    enabled_suggestion(P1, V1, P2, V2),
    available_version(P2, V),
    vc(ge, V, V2),
    in(P1, V1).

justified(P2, V) :-
    le_suggests(P1, V1, P2, V2),
    enabled_suggestion(P1, V1, P2, V2),
    vc(le, V, V2),
    available_version(P2, V),
    in(P1, V1).

%% We want to add recommended packages if possible.
in(P2) :-
    recommends(P1, V1, P2),
    enabled_recommendation(P1, V1, P2),
    in(P1, V1),
    not deny_recommendations,
    can_have(P2).

in(P2, V) :-
    gt_recommends(P1, V1, P2, V2),
    enabled_recommendation(P1, V1, P2, V2),
    available_version(P2, V),
    vc(gt, V, V2),
    in(P1, V1),
    not deny_recommendations,
    can_have(P2,V).

in(P2, V) :-
    lt_recommends(P1, V1, P2, V2),
    enabled_recommendation(P1, V1, P2, V2),
    vc(lt, V, V2),
    available_version(P2, V),
    in(P1, V1),
    not deny_recommendations,
    can_have(P2,V).

in(P2, V) :-
    ge_recommends(P1, V1, P2, V2),
    enabled_recommendation(P1, V1, P2, V2),
    vc(ge, V, V2),
```

```
    available_version(P2, V),
    in(P1, V1),
    not deny_recommendations,
    can_have(P2,V).

in(P2, V) :-
    le_recommends(P1, V1, P2, V2),
    enabled_recommendation(P1, V1, P2, V2),
    vc(le, V, V2),
    available_version(P2, V),
    in(P1, V1),
    not deny_recommendations,
    can_have(P2,V).

%% We want to add suggested packages if possible.
in(P2) :-
    suggests(P1, V1, P2),
    enabled_suggestion(P1, V1, P2),
    in(P1, V1),
    allow_suggestions,
    can_have(P2).

in(P2, V) :-
    gt_suggests(P1, V1, P2, V2),
    enabled_suggestion(P1, V1, P2, V2),
    vc(gt, V, V2),
    in(P1, V1),
    available_version(P2, V),
    allow_suggestions,
    can_have(P2,V).

in(P2, V) :-
    lt_suggests(P1, V1, P2, V2),
    enabled_suggestion(P1, V1, P2, V2),
    vc(lt, V, V2),
    in(P1, V1),
    available_version(P2, V),
    allow_suggestions,
    can_have(P2,V).

in(P2, V) :-
    ge_suggests(P1, V1, P2, V2),
    enabled_suggestion(P1, V1, P2, V2),
    vc(ge, V, V2),
    available_version(P2, V),
    in(P1, V1),
    available_version(P2, V),
    allow_suggestions,
    can_have(P2,V).
```

```
in(P2, V) :-
    le_suggests(P1, V1, P2, V2),
    enabled_suggestion(P1, V1, P2, V2),
    vc(le, V, V2),
    available_version(P2, V),
    in(P1, V1),
    allow_suggestions,
    can_have(P2,V).


%% We add standard and important packages if nobody conflicts with
%% them
in(P) :- standard(P), not cant_have(P).
in(P) :- important(P), not cant_have(P).

%% We cannot have a package if someone conflicts with it.
cant_have(P2) :- conflicts(P1, V1, P2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2).

cant_have(P2, V2) :-
    available_version(P2, V),
    vc(lt, V, V2),
    lt_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).

cant_have(P2, V2) :-
    available_version(P2, V),
    vc(gt, V, V2),
    gt_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).

cant_have(P2, V2) :-
    available_version(P2, V),
    vc(le, V, V2),
    le_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).

cant_have(P2, V2) :-
    available_version(P2, V),
    vc(ge, V, V2),
    ge_conflicts(P1, V1, P2, V2),
    in(P1, V1),
    enabled_conflict(P1, V1, P2, V2).
```

```
%% We can have a package that exists and is not conflicted.
can_have(P) :-
    package(P),
    not cant_have(P).

can_have(P,V) :-
    available_version(P, V),
    not cant_have(P,V).

%% If we cannot have a software package, we cannot have any concrete
%% versions of it.
cant_have(X,Y) :- available_version(X,Y), cant_have(X).

% Only one concrete version is allowed for each package.
cant_have(X,Y) :-
    1 { in(X,Z) : concrete_version(X,Z) : Z != Y },
    concrete_version(X,Y).

% We cannot have a virtual package without a provider.
cant_have(X) :- virtual(X), not has_provider(X).
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] Caldera OpenLinux. Available at: <url:http://www.calderasystems.com>.

[3] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30:232 – 282, June 1998.

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

[5] Debian FAQ. List of Frequently Asked Questions about Debian System. Available at: <url:http://www.debian.org/doc/FAQ/>.

[6] Debian packaging manual. Available at: <url:http://www.debian.org/doc/packaging-manuals/packaging.html/>.

[7] W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming,* 3:267–284, 1984.

[8] Gerhard Fleischanderl, Gerhard E. Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems & their applications*, pages 59 – 68, October 1998.

[9] Eugene C. Freuder. The role of configuration knowledge in the business process. *IEEE Intelligent Systems & their applications*, pages 29 – 31, October 1998.

[10] Gerhard Friedrich and Markus Stumptner. Consistency-based configuration. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05.* AAAI Press, 1999.

[11] Markus P.J. Fromherz. Model-based configuration of machine control software. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05.* AAAI Press, 1999.

[12] 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.

[13] GNU project. Website at: <url:http://www.gnu.org/>.

[14] Andreas Günter and Lothar Hotz. KONWERK - a domain independent configuration tool. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[15] Tudor Hulubei and Eugene C. Freuder. The goldilocks problem. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[16] U. John and U. Geske. ConBaCon – constraint based configuration. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[17] U. John and U. Geske. Reconfiguration of technical products using ConBaCon. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[18] Michael K. Johnson and Erik W. Troan. *Linux application development*. Addison Wesley Longman, Inc., 1998.

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

[20] Ravi Kapadia. Modeling for parametric system-level design optimization. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[21] W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38:588–619, 1991.

[22] Deborah L. McGuinness and Jon R. Wright. An industrial-strenght description logic-based configurator platform. *IEEE Intelligent Systems & their applications*, pages 69 – 77, October 1998.

[23] Tomi Männistö, Timo Soininen, Juha Tiihonen, and Reijo Sulonen. Framework and conceptual model for reconfiguration. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[24] Alexander Nareyek. Structural constraint satisfaction. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[25] I. Niemelä and P. Simons. Evaluating an algorithm for default reasoning. In *Working Notes of the IJCAI'95 Workshop on Applications and Implementations of Nonmonotonic Reasoning Systems, Montreal, Canada*, pages 66–72, Montreal, Canada, August 1995.

[26] I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, September 1996. The MIT Press.

[27] I. Niemelä, P. Simons, and T. Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 317–331, El Paso, Texas, USA, December 1999. Springer-Verlag.

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

[29] RedHat Linux. Available at: <url:http://www.redhat.com>.

[30] S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*. Prentice-Hall, Inc., 1995.

[31] 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.

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

[33] Mihaela Sabin and Eugene C. Freuder. Detecting and resolving inconsitency and redundancy in conditional constraint satisfaction problems. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[34] P. Simons. Efficient implementation of the stable model semantics for normal logic programs. Research Report 35, Helsinki University of Technology, Helsinki, Finland, September 1995.

[35] P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report 47, Helsinki University of Technology, Helsinki, Finland, August 1997.

[36] P. Simons. Extending the stable model semantics with more expressive rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 305–316, El Paso, Texas, USA, December 1999. Springer-Verlag.

[37] Slackware Linux. Available at: <url:http://www.cdrom.com>.

[38] Timo Soininen and Esther Gelle. Dynamic constraint satisfaction in configuration. In *Configuration Papers from the AAAI Workshop, AAAI Technical Report WS-99-05*. AAAI Press, 1999.

[39] Timo Soininen, Esther Gelle, and Ilkka Niemelä. A fixpoint definition of dynamic constraint satisfaction. In *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*. Springer-Verlag, October 1999.

[40] Timo Soininen and Niemelä Ilkka. Formalizing configuration knowledge using rules with choices. Technical Report TKO-B142, Laboratory of Information Processing Science, Helsinki University of Technology, 1998.

[41] 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.

[42] Bjarne Stroustrup. *The C++ Programming Language, third edition*. Addison-Wesley Publishing Company, 1997.

[43] Markus Stumptner and Franz Wotava. Model-based reconfiguration. In *Proceedings of the 5th Conference on Artificial Intelligence in Design '98*, 1998.

[44] S.u.S.E. Linux. Available at: <url:http://www.suse.com>.

[45] T. Syrjänen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B 18, Helsinki University of Technology, Helsinki, Finland, October 1998.

[46] W. F. Tichy. A data model for programmming support environments. In *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Information System Design and Development*, pages 31–48, January 1982.

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

[48] Andreas Zeller. *Configuration Management with Version Sets*. PhD thesis, Technische Universität Braunschweig, April 1997.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

HUT-TCS-A54	Antti Huima Analysis of Cryptographic Protocols via Symbolic State Space Enumeration. August 1999.
HUT-TCS-A53	Stefan Rönn Semantics of Semaphores. 1998.
HUT-TCS-A52	Ilkka Niemelä, Torsten Schaub (Eds.) Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning. May 1998.
HUT-TCS-A51	Kimmo Varpaaniemi On the Stubborn Set Method in Reduced State Space Generation. May 1998.
HUT-TCS-A50	Ilkka Niemelä (Ed.) Proceedings of the HeCSE Workshop on Emerging Technologies in Distributed Systems. March 1998.
HUT-TCS-A49	Tomi Janhunen Non-Monotonic Systems: A Framework for Analyzing Semantics and Structural Properties of NMR. March 1998.
HUT-TCS-A48	Tuomas Aura On the structure of delegation networks. December 1997.
HUT-TCS-A47	Patrik Simons Towards Constraint Satisfaction through Logic Programs and the Stable Model Semantics. August 1997.
HUT-TCS-A46	Tuomas Aura Stateless connections. May 1997.
HUT-TCS-A45	Keijo Heljanko Model Checking the Branching Time Temporal Logic CTL. May 1997.
HUT-TCS-A44	Karsten Schmidt Applying Reduction Rules to Algebraic Petri nets. March 1997.
HUT-TCS-A43	Kari J. Nurmela Minimum-Energy Point Charge Configurations on a Circular Disk. February 1997.
HUT-TCS-A42	Eero Lassila Towards Optimizing Code Generation by Domain-Sensitive Macro Expansion. January 1997.

ISBN 951-22-4881-6
ISSN 0783-5396