Appears in J. Loyd et al. (Eds.): CL 2000, LNAI 1861, 2000

Including Diagnostic Information in Configuration Models

Tommi Syrjänen*

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

Abstract. This work presents a new formal model for software configuration. The configuration knowledge is stored in a configuration model that is specified using a rule-based language. The language has a complete declarative semantics analogous to the stable model semantics for normal logic programs. In addition, a new method to add diagnostic information in configuration models is presented. The main idea is to divide the configuration process into two stages. At the first stage the user requirements are processed to check whether there exist any suitable configurations in the configuration model. In the second stage unsatisfiable requirements are diagnosed using a diagnostic model. The diagnostic model is constructed from the configuration model by adding a new set of atoms that represent the possible error conditions. The diagnostic output also explains why each problematic component was included in the configuration. As an example, a subset of the configuration problem for the Debian GNU/Linux system is formalized using the new rule-based language. Both configuration and diagnostic models of the problem are presented. The rule-language is implemented using an existing implementation of the stable model semantics, the Smodels system.

1 Introduction

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. There may also be a set of *constraints* imposed on the model that restrict the allowed object combinations. In a *configuration process*, we are given a configuration model and a set of *user requirements* and we want to find a configuration that satisfies the requirements. According to [12] configurations can be divided into three classes:

1. a *valid* configuration satisfies all constraints of the configuration model;
2. a *suitable* configuration is a valid configuration that also satisfies all user requirements; and

* This work has been supported by the Academy of Finland (project no. 43963) and Helsinki Graduate School in Computer Science and Engineering (HeCSE).

3. an *optimal* configuration is a suitable configuration that additionally satisfies some optimality criteria.

Most work that has been done in the configuration management field has focused on finding valid configurations and a recent survey on different configuration methods can be found in [6].

In this work we define a rule-based language RRL that can be used to express configuration knowledge. The RRL language allows the use of variables and it has a declarative semantics that is based on the stable model semantics for normal logic programs [3]. The main advantage of a declarative semantics is implementation-independence. In a configuration system without a well-defined semantics, configurations are defined by the behavior of the configuration tool and if the tool is changed the set of valid configurations of the system may also change. In addition, if there are no suitable configurations at all, we have to explain the user why this is the case. This explanation is difficult to do if the configuration model does not have a well-defined semantics.

One major aim of this work is to closely examine the situation where the user requirements cannot be satisfied. In these cases it is not enough to decide that a suitable configuration does not exist but we also need to give the user a diagnostic explanation that points out the problems. The diagnostic output should have at least the following properties:

1. the diagnosis should identify what components cause the problem;
2. for each component that is a part of the problem, the diagnosis should include an explanation why it was necessary to take it in the configuration; and
3. the diagnosis should be as concise as possible.

The reason for the second property is that the end-user may want to include a pair of components that are not mutually exclusive by themselves but that depend on a pair of conflicting components. If the justification information is not added to the diagnostic model, the user may have a difficult time trying to find out why the configuration system wants to add some seemingly unrelated components in the configuration.

The third property is important when configurations consist of thousands of components. If the diagnosis contains a lot of unessential information, the user will again have a difficult time trying to find the actual problem.

In this work we will take the approach that we will construct two distinct models for the configuration system. A *configuration model* encodes the components and the relationships between them and the diagnostic information is stored in a *diagnostic model*.

During a configuration task, we first try to find a configuration that satisfies the user requirements. If such configuration exists, we can configure the system according to it. Otherwise, we know that we cannot choose the components in such way that the resulting configuration is both valid and satisfies the user requirements and we have to compute a diagnosis. We can do this by constructing an invalid configuration that satisfies all user requirements and explaining why it violates the constraints of the model. This process in illustrated in Figure 1.

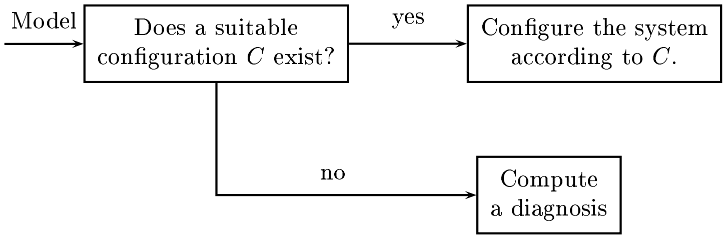

Fig. 1. A conceptual flowchart of a configuration task

The main reason why the two models are separated is simplicity. We find it easier to construct a configuration model when we do not have to worry about adding diagnostic information to it. Similarily, a diagnostic model is easier to construct when we do not have to worry about false alarms that are caused by incompatible optional components. Another reason for separation is efficiency. It seems that the current implementation is more efficient when the models are separated but the results are not conclusive and it is possible that there exists an efficient way to combine the data in one model.

As a practical example, we consider the configuration management problem of the Debian GNU/Linux system [1]. Debian is a distribution of the GNU/Linux operating system and currently with version 2.1 it has over 2500 distinct software packages. A package may interact with several other packages in various ways. A package may depend on functionality provided by another one, two packages may conflict with each other, or a package may recommend that another package is taken into a configuration whenever it is in it. The Debian distribution is an interesting case for configuration management for two main reasons:

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 large number of software packages makes the configuration management of the Debian system a non-trivial task and if a formal method can handle it well, it can probably also handle other difficult cases.

As it is not possible to model the configuration management of the Debian system completely in this work, we formalize only a small subset of it and concentrate on the diagnostic model. The whole system is modeled in [10] with some preliminary evaluation results.

2 The Rule Language

In this section, we introduce a declarative rule-based formal language RRL which is a subset of the language RL defined in [10]. The language RL is based on a configuration rule language presented in [9].

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

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

where p is a predicate symbol and a_1 to a_n are all variables or constants. We will use the convention that all variables start with a capital letter and all constants with a lower case letter. A *literal* is either an atom a or its negation $\mathrm{not}\, a$. 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 to each ground literal. If an atom a is in $\mathcal{V}$, 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 $\mathrm{not}\, a$ is *false* in $\mathcal{V}$ and vice versa.

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

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

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

where the atom h is the rule *head* and the literals $l_1, \ldots, l_n$ form the rule *body*. The rules of the form (2) are called *basic rules* and the rules of the form (3) are called *choice rules*. A basic rule with an empty rule body is called a *fact*. A *ground instance* of a rule is obtained by replacing all variables in it with constants. A RRL *program* is a set of rules.

The intuitive meaning of a basic rule is that if all literals in the rule body are true, the head atom also has to be true. If the body of a choice rule is true, we can include any subset of atoms h_1, ..., h_m in our model. Strictly speaking, choice rules are not necessary as they could be replaced by a adding a rule of the form:

$$\begin{aligned} h_i &\leftarrow l_1, \ldots, l_n, \mathrm{not}\, h'_i \\ h'_i &\leftarrow \mathrm{not}\, h_i \end{aligned} \tag{4}$$

for each $1 \leq i \leq m$. However, using choice rules makes programs more compact and the syntax is easy to expand to handle cases where a specific number of atoms h_1, ..., h_m has to be true when the body is true [5].

Next we define the formal semantics of RRL. The definition is very similar to the stable model semantics for normal logic programs [3]. The variables are handled by instantiating the rules with all possible constants that are used in the program and computing the stable models of the resulting program.

Definition 1. *The* Herbrand instantiation $\mathbf{HI}(P)$ *of a* RRL *program P is the set of all ground instances of rules in P that can be constructed using constant symbols in P.*

Example 1. Let $P = \{\mathsf{p(X)} \leftarrow \mathsf{q(X)};\ \mathsf{q(a)} \leftarrow;\ \mathsf{q(b)} \leftarrow\}$. The program $\mathbf{HI}(P)$ is now

$$\begin{aligned} \mathbf{HI}(P) = \{\ &\mathsf{p(a)} \leftarrow \mathsf{q(a)};\ \mathsf{p(b)} \leftarrow \mathsf{q(b)}; \\ &\mathsf{q(a)} \leftarrow;\ \mathsf{q(b)} \leftarrow \} \end{aligned} \tag{5}$$

Definition 2. *Given a ground* RRL *program P and a set of atoms M we construct the* reduct P^M *by*

1. *replacing all choice rules*

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

   with the set

$$\{h_i \leftarrow l_1, \ldots, l_n \mid 1 \leq i \leq m \wedge h_i \in M\} \tag{6}$$

   of rules.
2. *removing each rule that has a negative literal* $\mathrm{not}\, a$ *in its body where $a \in M$; and*
3. *removing all negative literals from the bodies of the remaining rules.*

Example 2. Let $P = \{\{a\} \leftarrow \mathrm{not}\, b, \{b\} \leftarrow \mathrm{not}\, a\}$ and $M_1 = \{a\}$. Now we can construct the reduct

$$P^{M_1} = \{a \leftarrow\}\ .$$

Similarly, if we set $M_2 = \emptyset$, the reduct $P^{M_2} = \emptyset$.

The reduct P^M is a set of Horn clauses so it has a unique minimal model $MM(P^M)$ [11]. If this minimal model coincides with M, we say that M is a *stable model* of P.

Definition 3. *Given a* RRL *program P and a set of atoms M, M is a* stable model *of P if and only if $M = MM(\mathbf{HI}(P)^M)$.*

Example 3. Let $P = \{\{a\} \leftarrow \mathrm{not}\, b, \{b\} \leftarrow \mathrm{not}\, a\}$. Now P has three stable models, $M_1 = \{a\}, M_2 = \{b\}$, and $M_3 = \emptyset$. As we saw in Example 2, the reduct of P with regards to M_1 is $P^{M_1} = \{a \leftarrow\}$ that has the minimal model $\{a\}$ which coincides to M_1. On the other hand, $M_4 = \{a, b\}$ is a model of P in a propositional sense but it is not a stable model as the reduct $P^{M_4} = \emptyset$ that has an empty minimal model.

Theorem 1. *Given a ground* RRL *program P the question whether there exists a stable model M of P is* NP*-complete.*

Proof. The NP-hardness follows directly from the fact that the problem of existence of a stable model for a normal logic program is NP-complete [4] and all normal logic programs are RRL programs.

If we guess the model M we can construct the reduct P^M in a linear time with respect to the number of literals in rule bodies and the minimal model $MM(P^M)$ can be computed in linear time [2]. This implies that the problem is in NP and thus it is NP-complete.

$$
\begin{aligned}
\{\ \mathsf{in}(\mathsf{P})\ \} &\leftarrow \mathsf{package}(\mathsf{P}), \mathsf{justified}(\mathsf{P}) \\
&\leftarrow \mathsf{in}(\mathsf{P}_1), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{not\ in}(\mathsf{P}_2) \\
&\leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2) \\
&\leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{not\ in}(\mathsf{P}) \\
&\leftarrow \mathsf{user\text{-}exclude}(\mathsf{P}), \mathsf{in}(\mathsf{P}) \\
\mathsf{justified}(\mathsf{P}) &\leftarrow \mathsf{user\text{-}include}(\mathsf{P}) \\
\mathsf{justified}(\mathsf{P}_2) &\leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \\
\mathsf{justified}(\mathsf{P}_2) &\leftarrow \mathsf{recommends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1)
\end{aligned}
$$

Fig. 2. The simplified Debian configuration model CM

3 Configuring a Debian System

In this section we formalize a small subset of the configuration management problem of the Debian GNU/Linux system. In particular, we will formalize only dependency, conflict, and recommendation relations of the system. We will leave out the package version information but we present the outline how it can be included in the model.

A package A *depends on* a package B if A cannot be used at all if B is not installed. A package A *conflicts with* B if A will not operate when B is installed on the system. A package A *recommends* B if B enhances the functionality of A in a significant way. The recommendation relation brings the concept of optionality to the configuration model. If a package recommends another, we can choose to add the recommended package or we can leave it out. The complete formalization of the Debian configuration system is presented in [10].

The Debian configuration model CM is constructed using the RRL language and it can be divided into two parts:

1. a database that stores information about the packages and their relations as facts; and
2. a set of inference rules that construct the valid configurations using the facts stored in the database.

The software packages are modeled as constants in the database. For each package P we add a corresponding constant p to the program. The relations are modeled with predicates, for example, if a package P_1 depends on P_2, the atom $\mathsf{depends}(\mathsf{P}_1,\ \mathsf{P}_2)$ is added as a fact to the program. The inference rules that form the core of the configuration model are presented in Figure 2.

As a Debian configuration is essentially a set of packages, we will use the predicate $\mathsf{in}(\mathsf{P})$ to denote that the package P is chosen to be in the configuration. Thus, valid configurations of a configuration model correspond to the stable models of the RRL program.

The user requirements can be modeled using two predicates $\mathsf{user\text{-}include(P)}$ and $\mathsf{user\text{-}exclude(P)}$. A set U of user requirements is a set of facts of the form

$$\begin{array}{ll}\mathsf{user\text{-}include(P)} \leftarrow & \text{and} \\ \mathsf{user\text{-}exclude(P)} \leftarrow & \end{array}$$

where the atom $\mathsf{user\text{-}include(P)} \in U$ if the user explicitly selected P to be in the configuration and $\mathsf{user\text{-}exclude(P)} \in U$ when the user wants to ensure that P is not in the configuration.

Definition 4. *Given a Debian configuration model CM, a set U of user requirements, and a stable model M of $CM \cup U$, a Debian* configuration C_M *is the set of packages*

$$C_M = \{\mathsf{P} \mid \mathsf{in(P)} \in M\} \ . \tag{7}$$

We want the configurations to be compact. They should contain the user selected packages and the packages they depend on and nothing more. Additionally, if a package is recommended by some package in the configuration, it may be included in the configuration but it may also be left out. We encode this principle by using the predicate $\mathsf{justified(P)}$ which is true in a stable model if the package P has some reason to be in a configuration.

The basic rule of the configuration model is that a package may be added to a configuration if and only if it can be justified:

$$\{\ \mathsf{in(P)}\ \} \leftarrow \mathsf{justified(P)}, \mathsf{package(P)} \ . \tag{8}$$

The predicate $\mathsf{package(P)}$ is added to the rule body to ensure that only available packages may be added to the configuration.

It is an error if a package P_1 is in a configuration but one of its dependencies is not satisfied:

$$\leftarrow \mathsf{in(P_1)}, \mathsf{depends(P_1, P_2)}, \mathsf{not\ in(P_2)} \ . \tag{9}$$

We may not have two conflicting packages in one configuration:

$$\leftarrow \mathsf{conflicts(P_1, P_2)}, \mathsf{in(P_1)}, \mathsf{in(P_2)} \ . \tag{10}$$

If the user made an explicit choice regarding to a package P, the choice must be adhered:

$$\leftarrow \mathsf{user\text{-}include(P)}, \mathsf{not\ in(P)} \tag{11}$$
$$\leftarrow \mathsf{user\text{-}exclude(P)}, \mathsf{in(P)} \ . \tag{12}$$

A package is justified if either the user selected it, some package that has to be in the configuration depends on it, or some included package recommends it.

$$\mathsf{justified(P)} \leftarrow \mathsf{user\text{-}include(P)} \tag{13}$$
$$\mathsf{justified(P_2)} \leftarrow \mathsf{depends(P_1, P_2)}, \mathsf{in(P_1)} \tag{14}$$
$$\mathsf{justified(P_2)} \leftarrow \mathsf{recommends(P_1, P_2)}, \mathsf{in(P_1)} \ . \tag{15}$$

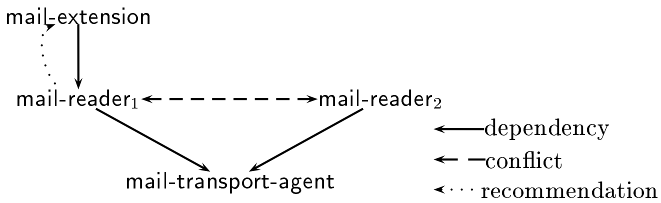

Fig. 3. The relationships between packages in Example 4

As an example we consider a simple system that consists of four different packages. There are two different mail reader packages (mail-reader$_1$ and mail-reader$_2$) that are mutually exclusive and both of them need an installed mail transport agent (mail-transport-agent) package to work correctly. In addition, mail-reader$_1$ recommends an extension package mail-extension that does not work without mail-reader$_1$. The relationships between the packages are shown in Figure 3 and the facts that encode the relationships are presented in Table 1.

Example 4. Suppose that the user selects the mail-reader$_1$ package by adding the atom user-include(mail-reader$_1$) as a fact to the program.

We can conclude by rules (13) and (11) that atoms justified(mail-reader$_1$) and in(mail-reader$_1$) have to be in the stable model of the program. As mail-reader$_1$ has to be in a configuration, we can justify mail-transport-agent by (14) and mail-extension by (15). The last package cannot be justified. We now have two justified packages that we can include in the configuration by the rule (8). The rule (9) forces us to add in(mail-transport-agent) to the model. The only choice we have left is whether to add the mail-extension package or not. Thus, we have two suitable configurations C_1 and C_2.

$$C_1 = \{ \ \mathsf{in}(\mathsf{mail\text{-}reader}_1), \mathsf{in}(\mathsf{mail\text{-}transport\text{-}agent}), \mathsf{in}(\mathsf{mail\text{-}extension}) \ \}$$
$$C_2 = \{ \ \mathsf{in}(\mathsf{mail\text{-}reader}_1), \mathsf{in}(\mathsf{mail\text{-}transport\text{-}agent}) \ \}$$

Example 5. Suppose that the user decided to have the mail-extension package and wants to use it with mail-reader$_2$. These requirements are modeled by adding facts user-include(mail-extension) and user-include(mail-reader$_2$) to the program. Now, by (14) the atom in(mail-reader$_1$) has to be in the model which leads to

Table 1. The facts used to encode Example 4.

package(mail-reader$_1$)	depends(mail-reader$_1$, mail-transport-agent)
package(mail-reader$_2$)	depends(mail-reader$_2$, mail-transport-agent)
package(mail-extension)	depends(mail-extension, mail-reader$_1$)
package(mail-transport-agent)	conflicts(mail-reader$_1$, mail-reader$_2$)
recommends(mail-reader$_1$, mail-extension)	

a contradiction with $\mathsf{in}(\mathsf{mail\text{-}reader}_2)$ by (10). As a consequence, we do not have any suitable configurations.

The main weakness of the configuration model presented above is that it does not model the version information of the system. This would be unacceptable in real applications so we need to find a way to extend the model. We present here only a brief outline of the extension and the details can be found in [10].

We associate different versions of a package together using the predicate available-version(P, V) that is true when a version V of a package P is present in the configuration model. The predicate in is divided into two cases:

1. an atom in(P, V) is true exactly when a version V of P is chosen to be in the configuration; and
2. an atom in(P) is true when any version of P is in the configuration.

The relationships between packages may be parametrized with version numbers. For example, a package P may depend on Q version V or later. These kinds of dependencies can be modeled using atoms of the form depends(P, Q, Op, V) where Op is the corresponding relational operator. We then add rules that ensure that at least one compatible version of Q is in the configuration when P is in it. These rules may be expressed compactly by extending the language to allow *cardinality literals* of the form:

$$L \leq \{l_1, \ldots, l_n\} \leq U \tag{16}$$

where L and U are integral lower and upper bounds, respectively, and l_1, ..., l_n are literals. The intuition of a cardinality literal is that a cardinality literal is satisfied when the number of satisfied literals l_1, ..., l_n is between L and U, inclusive. Using cardinality literals the dependency constraint can be expressed as:

$$\begin{aligned} \leftarrow\ & 0 \leq \{\mathsf{in}(\mathsf{P}_2,\ \mathsf{V}) : \mathsf{available\text{-}version}(\mathsf{P}_2,\ \mathsf{V}) : \mathsf{V} \geq \mathsf{V}_2\} \leq 0, \\ & \mathsf{depends}(\mathsf{P}_1,\ \mathsf{P}_2,\ \geq,\ \mathsf{V}_2), \\ & \mathsf{in}(\mathsf{P}_1) \end{aligned} \tag{17}$$

where $\mathsf{in}(\mathsf{P}_2,\ \mathsf{V}) : \mathsf{available\text{-}version}(\mathsf{P}_2,\ \mathsf{V})\text{: } \mathsf{V} \geq \mathsf{V}_2$ denotes the set of packages that may potentially satisfy the dependency. This set can be automatically computed during instantiation of the logic program.

4 The Diagnostic Model

We will construct the diagnostic model DM by adding a new set of atoms that represent the potential error conditions and explanations. We modify the constraints in such way that the diagnostic model will always have at least one stable model. The new atoms that are true in a stable model will then identify a set of errors in the requirements and give an explanation why this is the case.

The new atoms can be divided into three classes:

$$
\begin{aligned}
\mathsf{in}(\mathsf{P}_2) &\leftarrow \mathsf{in}(\mathsf{P}_1), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{package}(\mathsf{P}_2) \\
\mathsf{missing}(\mathsf{P}_2) &\leftarrow \mathsf{in}(\mathsf{P}_1), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{not\ package}(\mathsf{P}_2) \\
\mathsf{in}(\mathsf{P}) &\leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{package}(\mathsf{P}) \\
\mathsf{missing}(\mathsf{P}) &\leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{not\ package}(\mathsf{P}) \\
\mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2) &\leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2) \\
\mathsf{in\text{-}conflict}(\mathsf{P}, \mathsf{user\text{-}exclude}) &\leftarrow \mathsf{in}(\mathsf{P}), \mathsf{user\text{-}exclude}(\mathsf{P}) \\
\mathsf{needs\text{-}reason}(\mathsf{P}) &\leftarrow \mathsf{missing}(\mathsf{P}) \\
\mathsf{needs\text{-}reason}(\mathsf{P}_1) &\leftarrow \mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{package}(\mathsf{P}_1) \\
\mathsf{needs\text{-}reason}(\mathsf{P}_2) &\leftarrow \mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{package}(\mathsf{P}_2) \\
\mathsf{needs\text{-}reason}(\mathsf{P}_1) &\leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{needs\text{-}reason}(\mathsf{P}_2), \mathsf{in}(\mathsf{P}_1) \\
\mathsf{user\text{-}selected}(\mathsf{P}) &\leftarrow \mathsf{needs\text{-}reason}(\mathsf{P}), \mathsf{user\text{-}include}(\mathsf{P}) \\
\mathsf{needs}(\mathsf{P}_1, \mathsf{P}_2) &\leftarrow \mathsf{needs\text{-}reason}(\mathsf{P}_2), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1)
\end{aligned}
$$

Fig. 4. The simplified Debian diagnostic model DM

1. Atoms that denote error conditions. We will use the predicates $\mathsf{missing}(\mathsf{P})$ and $\mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2)$ for this purpose.
2. Atoms that mark the packages that are in some way part of the problem and thus need an explanation. The predicate $\mathsf{needs\text{-}reason}(\mathsf{P})$ is used for this.
3. Atoms that explain why certain packages were taken into the configuration. The predicates $\mathsf{user\text{-}selected}(\mathsf{P})$ and $\mathsf{needs}(\mathsf{P}_1, \mathsf{P}_2)$ are used for this purpose.

Using these predicates we can define a Debian diagnosis formally.

Definition 5. *Given a Debian diagnostic model DM and a set U of user requirements, a* diagnosis *D is a four-tuple $D = (M, E_M, P_M, R_M)$, where*

1. M is a stable model of $DM \cup U$;
2. E_M is the error set

$$E_M = \{\mathsf{missing}(\mathsf{P}) \in M\} \cup \{\mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2) \in M\} \ ; \tag{18}$$

3. P_M is the problem set

$$P_M = \{\mathsf{P} \mid \mathsf{needs\text{-}reason}(\mathsf{P}) \in M\} \ ; \ \textit{and} \tag{19}$$

4. R_M is the explanation set

$$R_M = \{\mathsf{user\text{-}selected}(\mathsf{P}) \in M\} \cup \{\mathsf{needs}(\mathsf{P}_1, \mathsf{P}_2) \in M\} \ . \tag{20}$$

The new program may have more than one diagnosis if there are more than one way to break the constraints. Each diagnosis corresponds to a set of choices that lead to a contradiction. When diagnosing a problem it is often enough to examine only one model. The reason for this is that if all possible choices that we can make lead to a contradiction, it does not matter much what particular set of choices we examine. However, the diagnosis should prefer conflicts over missing packages. The following example shows why this is necessary.

Example 6. 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 to work but both C and D conflict with B.

There are three ways to invalidate the constraints. If we add C in the configuration, it is in 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 real problem is that including both packages A and B in same configuration leads to a conflict. In the third case we only notice that A does not have its dependencies satisfied but we cannot directly see what caused the problem.

The first modification to the configuration model is that we remove the rule (8) because we want the diagnosis to contain only those packages that have to be in there so that it will be as small as possible. In addition, this ensures that no false alarms are caused by adding unnecessary recommended packages in the configuration.

4.1 Missing Packages

We will use the atom missing(P) to denote that some package in the configuration depends on P but for some reason P is not in the configuration. As we take the approach that we add all necessary packages to a configuration, a package may be missing only if it is not present in the configuration model at all. We replace the rule (9) of the configuration model with the pair of rules

$$\mathsf{in}(\mathsf{P}_2) \leftarrow \mathsf{in}(\mathsf{P}_1), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{package}(\mathsf{P}_2) \tag{21}$$

$$\mathsf{missing}(\mathsf{P}_2) \leftarrow \mathsf{in}(\mathsf{P}_1), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{not\ package}(\mathsf{P}_2) \ . \tag{22}$$

The rule (21) ensures that existing packages are added to the model and the rule (22) marks non-existing packages as missing. In addition, it may be the case that a package the user explicitly included in the configuration is not available. To handle this situation we replace the rule (11) with rules

$$\mathsf{in}(\mathsf{P}) \leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{package}(\mathsf{P}) \tag{23}$$

$$\mathsf{missing}(\mathsf{P}) \leftarrow \mathsf{user\text{-}include}(\mathsf{P}), \mathsf{not\ package}(\mathsf{P}) \ . \tag{24}$$

4.2 Conflicts

We will use the predicate in-conflict to model conflicts. We define the atom in-conflict(P_1, P_2) to be true exactly when P_1 and P_2 conflict with each other and they both have to be in the configuration.

In the diagnostic model we replace the rule (10) with

$$\mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2) \leftarrow \mathsf{conflicts}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1), \mathsf{in}(\mathsf{P}_2) \ . \tag{25}$$

We must also handle the case where the user wants to leave out a package that some other package needs. We will use a special constant $\mathsf{user\text{-}exclude}$ and the rule

$$\mathsf{in\text{-}conflict}(\mathsf{P}, \mathsf{user\text{-}exclude}) \leftarrow \mathsf{in}(\mathsf{P}), \mathsf{user\text{-}exclude}(\mathsf{P}) \tag{26}$$

to model these cases.

4.3 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 the user chose it to be in the configuration. A package is also necessary if some mandatory package depends on it.

We will use the predicate $\mathsf{needs\text{-}reason}$ to mark the atoms that we want to justify. We can find the explanations by first marking the packages that directly cause the problems and then recursively marking all packages that depend on marked packages and are present in the configuration. This can be accomplished with four rules:

$$\mathsf{needs\text{-}reason}(\mathsf{P}) \leftarrow \mathsf{missing}(\mathsf{P}) \tag{27}$$

$$\mathsf{needs\text{-}reason}(\mathsf{P}_1) \leftarrow \mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{package}(\mathsf{P}_1) \tag{28}$$

$$\mathsf{needs\text{-}reason}(\mathsf{P}_2) \leftarrow \mathsf{in\text{-}conflict}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{package}(\mathsf{P}_2) \tag{29}$$

$$\mathsf{needs\text{-}reason}(\mathsf{P}_1) \leftarrow \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{needs\text{-}reason}(\mathsf{P}_2), \mathsf{in}(\mathsf{P}_1)\ . \tag{30}$$

We model the justifications using two predicates, $\mathsf{user\text{-}selected}$ and needs. The atom $\mathsf{user\text{-}selected}(\mathsf{P})$ is true when the user chose P to be in the configuration and the atom $\mathsf{needs}(\mathsf{P}_1, \mathsf{P}_2)$ is true if P_2 was included in the configuration because P_1 depends on it. The justifications can be modeled with the following two rules:

$$\mathsf{user\text{-}selected}(\mathsf{P}) \leftarrow \mathsf{needs\text{-}reason}(\mathsf{P}), \mathsf{user\text{-}include}(\mathsf{P}) \tag{31}$$

$$\mathsf{needs}(\mathsf{P}_1, \mathsf{P}_2) \leftarrow \mathsf{needs\text{-}reason}(\mathsf{P}_2), \mathsf{depends}(\mathsf{P}_1, \mathsf{P}_2), \mathsf{in}(\mathsf{P}_1)\ . \tag{32}$$

Example 7. Reconsider the situation that was presented in Example 5. By the rule (23) we have to include both atoms $\mathsf{in}(\mathsf{mail\text{-}reader}_2)$ and $\mathsf{in}(\mathsf{mail\text{-}extension})$ in the model and (21) adds atoms $\mathsf{in}(\mathsf{mail\text{-}reader}_1)$ and $\mathsf{in}(\mathsf{mail\text{-}transport\text{-}agent})$. By (25) we include the atom $\mathsf{in\text{-}conflict}(\mathsf{mail\text{-}reader}_1,\ \mathsf{mail\text{-}reader}_2)$ in the model.

As we have a conflict, we want to find the reason for it. By rules (28) and (29) we know that $\mathsf{needs\text{-}reason}(\mathsf{mail\text{-}reader}_1)$ and $\mathsf{needs\text{-}reason}(\mathsf{mail\text{-}reader}_2)$ are in the model. Using the rule (30) we see that we have to find a reason for $\mathsf{mail\text{-}extension}$, also.

By using the rule (31), we find explanations $\mathsf{user\text{-}selected}(\mathsf{mail\text{-}extension})$ and $\mathsf{user\text{-}selected}(\mathsf{mail\text{-}reader}_2)$. We get the justification for the last package by using the rule (32) to find that $\mathsf{needs}(\mathsf{mail\text{-}extension},\ \mathsf{mail\text{-}reader}_2)$ has to be in the model. Nothing else has to be included in the model so we now have the full diagnosis:

$$\begin{aligned}
E &= \{\mathsf{in\text{-}conflict}(\mathsf{mail\text{-}reader}_1,\ \mathsf{mail\text{-}reader}_2)\} \\
P &= \{\mathsf{mail\text{-}reader}_1, \mathsf{mail\text{-}reader}_2, \mathsf{mail\text{-}extension}\} \\
R &= \{\mathsf{user\text{-}selected}(\mathsf{mail\text{-}reader}_2), \mathsf{user\text{-}selected}(\mathsf{mail\text{-}extension}), \\
&\quad\ \mathsf{needs}(\mathsf{mail\text{-}extension},\ \mathsf{mail\text{-}reader}_1)\} \ .
\end{aligned}$$

5 Implementation

The configuration and diagnostic models have been implemented as extended logic programs using the SMODELS system [7, 8] developed in the Laboratory for Theoretical Computer Science in Helsinki University of Technology. The smodels system is available at

`<url:http://www.tcs.hut.fi/pub/smodels>` .

Both the models described in this work and in [10] are available at

`<url:http://www.tcs.hut.fi/%7Etssyrjan/configuration/>` .

There have been some preliminary tests on the full Debian configuration model and the results are presented in [10]. The model was generated using actual data from Debian version 2.1 with 2260 different packages. The model was tested by generating a random set of user requirements and measuring the time that was used to either find a suitable configuration or a diagnosis. The tests were run on a 233 MHz Intel Pentium II with 128 MB of main memory. The compilation of the model took approximately 13 seconds and after that a valid configuration could be found in about 2 seconds. The results were similar for the diagnostic model.

The models are not yet incorporated into a concrete configuration tool but the test results give hope that it would be possible to use this approach also in practice.

6 Conclusions and Future Work

We presented a method to add diagnostic information to configuration models that are defined using a rule-based language RRL. The approach was to divide the configuration task into two phases. In the first phase we try to find a configuration that satisfies the user requirements. If one is found, the configuration task is completed. Otherwise, in the second phase we diagnose the user requirements trying to find an explanation why they are unsatisfiable.

We generate the diagnostic model by adding a new set of atoms to the configuration model to represent the possible error conditions and modify the rules to ensure the diagnostic model has always at least one stable model whatever the user requirements are. The new atoms that are true in the diagnosis identify a set of errors in the requirements. In addition, the diagnosis also contains explanations that tell for each component that is related to an error the reason why it has to be in the configuration.

We modeled a part of the configuration management problem of the Debian GNU/Linux system using RRL. Both configuration and diagnostic models were constructed. The configuration objects were distinct software packages that may depend on, conflict with, or recommend each other. The models were simplified and the problem of version management was addressed only briefly.

There were two possible error types for user requirements in the Debian configuration model. The first one was that necessary packages might be missing from the configuration. The second one was that two conflicting packages might be in a configuration. We defined the diagnostic model in such way that conflicts were preferred over missing packages because they give more information to the user: if a package is left out because it otherwise would cause a conflict, the user would notice only the absence of the package but the real cause would not be apparent.

The explanations are generated by marking all packages that are missing or in conflict with another one as problematic packages. In addition, all packages that depend on the problematic packages are marked, too. This is done because it is possible that the problems arise because some otherwise unrelated packages depend on conflicting packages.

The full configuration model for the Debian system is presented in [10]. The model has not been incorporated into any existing tool and the next step of this research is to construct a simple back-end that can be used for further testing in real environment. If the results are still promising, the model can be integrated with an existing Debian configuration tool.

References

1. Debian GNU/Linux. Available at: <url:http://www.debian.org>.
2. 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.
3. 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.
4. W. Marek and M. Truszczyński. Autoepistemic logic. *Journal of the Association for Computing Machinery*, 38:588–619, 1991.
5. Ilkka Niemelä, Patrik Simons, and Timo Soininen. Stable model semantics of weight constraint rules. In *Proceedings of the Fifth Interational Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, December 1999.
6. Daniel Sabin and Rainer Weigel. Product configuration frameworks — a survey. *IEEE Intelligent Systems & their applications*, pages 42 – 49, October 1998.
7. P. Simons. Efficient implementation of the stable model semantics for normal logic programs. Research Report 35, Helsinki University of Technology, Helsinki, Finland, September 1995.

8. 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.
9. Timo Soininen and Ilkka Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*. Springer-Verlag, January 1999.
10. T. Syrjänen. A rule-based formal model of software configuration. Research Report A 55, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Helsinki, Finland, December 1999.
11. M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23:733–742, 1976.
12. B. Wielinga and G. Schreiber. Configuration-design problem solving. *IEEE Expert*, 12 2:49 – 56, March-April 1997.