Version Spaces and Rule-Based Configuration Management

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

We develop a rule-based method for representing version spaces of software products. The version information is expressed with version clauses that may be combined using conjunction and disjunction. A configuration model is divided into a database and a set of inference rules. The rules have a declarative semantics and they are implemented using the extended rule types of the SMODELS system.

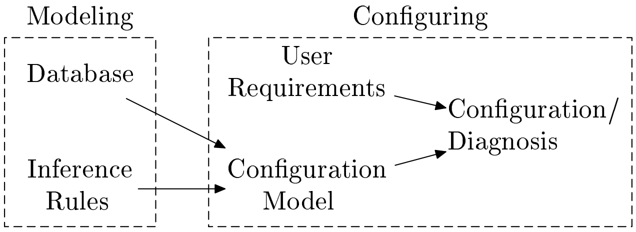

Figure 1: The conceptual flow of a configuration process

1 Introduction

A *configuration model* consists of a set *Comp* of *components* and a set *Const* of *constraints*. Each component is of the form c-v, where c is the component *base name* and v is the *version string*. A *configuration* is a subset $\mathcal{C} \subseteq Comp$ that satisfies all constraints. In a configuration problem we are given a set of user requirements and we want to construct a configuration that satisfies the requirements.

The configuration problem is difficult even when the underlying system stays constant, and in practice that happens only rarely. If a configurable product is successful, it will evolve with time; new components are added and old components are modified so the configuration model has to be changed accordingly.

In this work we consider the problem of configuring large software products that have possibly hundreds or thousands of components that each may have several different versions. Maintaining this kind of a configuration system is not easy. We try to create a formal model that can be updated with minimal user effort when the product changes. The basic idea is to divide the configuration model into two separate parts:

- *configuration database* that contains the knowledge on components and their versions; and
- a set of *inference rules* that act on the database.

The inference rules are defined only once, when the product is first modeled, and after that they remain unchanged. The database should contain only simple facts so that its maintenance is as easy as possible.

The database and inference rules are then combined into the actual configuration model that is used in a stand-alone tool. The user requirements are given to the tool as input and it then constructs either a suitable configuration or a diagnosis that explains why the requirements are unsatisfiable [Syrjänen, 2000a]. This process is illustrated in Figure 1.

Our main contribution is a set of inference rules that implement the dependency, incompatibility, and choice constraints [Soininen and Niemelä, 1999] for configuration systems with complex version spaces. A component a *depends* on a component b if a does not function correctly if b is not in the configuration. Components a and b are *incompatible*, or conflict with each other, if both cannot be in the configuration at the same time. The choices are used to encode optionality in configurations. For example, we may want to express that after including the component a in the configuration, we may include one or more of b, c, and d.

The constraints are defined in terms of version clauses that represent sets of different versions of components. With version clauses we can express, for example, the following types of relations:

- The component A depends on B version 2.0 or later;
- The component A conflicts with the 1.5 variant branch of B;
- If A is in a configuration, we may add exactly one version of either B or C to it.

The inference rules have a formal declarative semantics and they are implemented using the extended rule types of the SMODELS system [Niemelä *et al.*, 2000].

2 Version Spaces

The configuration knowledge can be divided into two separate dimensions, the *product space* and the *version space*

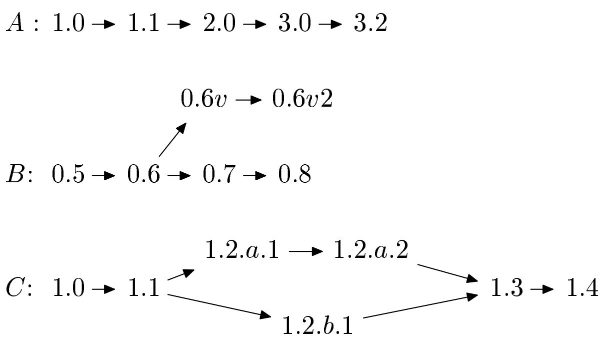

Figure 2: Simple examples on version graphs

[Conradi and Westfechtel, 1998]. The configuration components and their relationships form the product space and the version space describes the relations of different versions of components. The two spaces are not completely orthogonal since the relationships between two components may depend on their specific versions.

A version space is most often represented as a graph where the different version identifiers form the nodes and there is an edge between nodes a and b if b is a direct modification of a. The difference between b and a is called their *delta*. If we intend that a new version should replace the older one, we call it a *revision*. Conversely, if we want to have both versions available, we call them *variants*.

The Figure 2 contains three simple version graphs. The component A has a linear version history and a new version has always replaced the old one. The component B has two *branches* that have diverged at version 0.6. The component C has also two branches but this time the branches have been *merged* again at version 1.3.

2.1 Expressing Version Relationships

We now define a simple language for expressing version relationships. The basic building block is a *version clause* of the form $(c\ op\ v)$ where c is a component name, v is a version string and $op \in \{<, \leq, =, \neq, \geq, >\}$ is a relational operator. A basic version clause A denotes the set $V(A)$ of versions v' of c for which $v'\ op\ v$ holds. Note that the relation is defined by arcs of the version graph with $v' < v$ if there is a path from v' to v in the graph. We use two special version strings, *first* and *current* to denote the first and the latest versions of the components.

We also define two other simple clauses[1]:

- $(\mathit{any}\ c)$ denotes all versions of c; and
- $(c\ \mathit{branch}\ b)$ denotes the whole branch b of c.

We may combine version clauses using connectives $\wedge$ and $\vee$. If A and B are clauses, $(A \wedge B)$ denotes the intersection $V(A) \cap V(B)$. A disjunction $(A \vee B)$ is defined differently and it denotes the set $\{V(A), V(B)\}$. The intuition here is that a disjunction models a choice over sets of components while a conjunction specifies what specific versions of a component should belong to one set. The compound version clauses have to be in disjunctive normal form. That is, a compound clause is of the form:

$$(A_1 \wedge A_2 \cdots \wedge A_n) \vee \cdots \vee (B_1 \wedge B_2 \wedge \cdots \wedge B_m)\ . \quad (1)$$

In some cases, for example when examining dependencies, we want to check whether a version clause is *satisfied* in a configuration or not. In the basic case a clause is satisfied if some component version that belongs to it is in a configuration. However, we may want to express more complex choices. For this, we define an integral u/l-cardinality for each version clause A. This is denoted by A_l^u. The intuition is that the clause is satisfied if there are between l and u components from it in the configuration, or in the case of a disjunction that between l and u disjuncts should be satisfied. If we do not want to impose a cardinality on a clause, we can leave it out. In that case it is interpreted as an $\infty/1$-cardinality.

Definition 2.1 *Let A_l^u be a version clause and $\mathcal{C}$ be a configuration. Then, A_l^u is* satisfied *in $\mathcal{C}$ if one of the following conditions hold:*

1. *A_l^u is a simple clause or a conjunction of the form $(A_1 \wedge \cdots \wedge A_n)_l^u$, and*
$$l \leq |\mathcal{C} \cap V(A)| \leq u\ .$$
2. *A_l^u is a disjunction of the form $(A_1 \vee \cdots A_n)_l^u$, $S_A = \{A_i \mid A_i$ is satisfied in $\mathcal{C}\}$ and*
$$l \leq |S| \leq u\ .$$

The satisfiability of a disjunctive clause is well-defined because the clauses are in disjunctive normal form.

Example 2.1 *Consider the version space shown in Figure 2 and the following two version clauses:*

$$C_1 : ((A \geq 1) \wedge (A < 2))$$
$$C_2 : ((B > 0.7) \vee (C < 1.1))_1^1$$

Now

$$V(C_1) = \{A\text{-}1.0, A\text{-}1.1\}$$
$$V(C_2) = \{\{B\text{-}0.8\}, \{C\text{-}1.0\}\}$$

Consider the configuration $\mathcal{C} = \{A\text{-}1.1, C\text{-}1.0, B\text{-}0.8\}$. The clause C_1 is satisfied since $A\text{-}1.1 \in \mathcal{C} \cap V(C_1)$. The clause C_2 is not satisfied, since the number of its satisfied subclauses is $2 \geq 1$ that was the upper bound.

3 Rule Language

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

$$p(a_1, \ldots, a_n)\ , \quad (2)$$

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 a atom a, its negation $\mathrm{not}\ a$, or a *cardinality constraint* of the form:

$$L \leq \{a_1, \ldots, a_n, \mathrm{not}\ b_1, \ldots, \mathrm{not}\ b_m\} \leq U\ , \quad (3)$$

[1] It would also be possible to define dual clauses $(\mathit{none}\ c)$ and $(c\ \mathit{not\text{-}in\text{-}branch}\ b)$ but they are left out for brevity.

where $a_1, \ldots a_n$, $b_1, \ldots b_m$ are atoms and L, U are integral *lower* and *upper bounds*. The bounds may also be left out. In that case we set the lower bound to 0 and upper bound to infinity. A literal that is not a cardinality constraint is a *basic literal*. A literal is *ground* if it does not have any variables. We may use the notation $p(X) : q(X)$ to denote the set of atoms $p(X)$ for which $q(X)$ is also true.

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

$$h \leftarrow l_1, \ldots, l_n \ , \tag{4}$$

where the literal h is the rule *head* and the literals $l_1, \ldots, l_n$ form the rule *body*. If h is an atom, the rule is a *basic rule*, otherwise it is an *extended rule*. A basic rule with an empty rule body is called a *fact*. A *program* is a set of rules. A rule is *ground* if all literals in it are ground. A rule with variables denotes the set of ground rules that can be obtained by substituting the variables with constants of the program.

We are interested in finding answer sets of rule programs. Here we give only quite informal definition for them. A formal one can be found in [Niemelä and Simons, 2000]. Intuitively, an answer set is a set of atoms that satisfies all rules of the program and each atom in it is justified in the sense that it occurs in a rule head that has its body satisfied by the set. An atom a is satisfied in an answer set M if $a \in M$. Respectively, a literal not a is satisfied if $a \notin M$. A cardinality constraint $L \leq \{l_1, \ldots, l_n\} \leq U$ is satisfied if the number of satisfied literals $l_1, \ldots, l_n$ is between L and U, inclusive. A rule $h \leftarrow l_1, \ldots, l_n$ is satisfied if either one of the literals l_i in its body is not satisfied or h is satisfied.

Example 3.1 *Let P be the program:*

$$\begin{aligned} a &\leftarrow \text{not } b \\ b &\leftarrow \text{not } a \\ c &\leftarrow 2 \leq \{a, d\} \end{aligned}$$

Then P has two answer sets, $M_1 = \{a\}$ and $M_2 = \{b\}$. Let us consider M_1. The first two rules are satisfied since $a \in M_1$. The body of the third rule is not satisfied because only one atom of $\{a, d\}$ is in M_1 so we do not have to add c into the answer set.

Example 3.2 *Let P be the program:*

$$\begin{aligned} a(1) &\leftarrow \\ a(2) &\leftarrow \\ 1 \leq \{b(X,Y) : a(Y)\} \leq 1 &\leftarrow a(X) \end{aligned}$$

Then the third rule denotes the set:

$$\begin{aligned} 1 \leq \{b(1,1), b(1,2)\} \leq 1 &\leftarrow a(1) \\ 1 \leq \{b(2,1), b(2,2)\} \leq 1 &\leftarrow a(2) \end{aligned}$$

of rules. Since both $a(1)$ and $a(2)$ are true, P has four answer sets[2]: $M_1 = \{b(1,1), b(2,1)\}$, $M_2 = \{b(1,1), b(2,2)\}$, $M_3 = \{b(1,2), b(2,1)\}$, $M_4 = \{b(1,2), b(2,1)\}$.

[2] The atoms $a(1)$ and $a(2)$ are left out of the answer sets for clarity.

4 Encoding Version Spaces

In this section we define a rule-based representation of version spaces. First we define encodings for components and version graphs and then we show how they can be used to encode the version clauses.

4.1 Components

The components and their versions are modeled separately using constants. For example, the component a-1.0 generates the constants a and 1.0 to the model. The first one is the component base name and the second the version string.

A version string is connected to the corresponding base name using the predicate $\mathit{component}(\mathit{name}, \mathit{version})$. As we do not want to worry about old components, we use the predicate $\mathit{available\text{-}component}(\mathit{name}, \mathit{version})$ to denote that $\mathit{name\text{-}version}$ is still available.

We use the predicate $\mathit{in}(c, v)$ to denote that the component c-v is in the configuration.

4.2 Version Graph

We encode the version graph using the predicate $\mathit{delta}/4$. The fact:

$$\mathit{delta}(c_1, v_1, c_2, v_2) \leftarrow \tag{5}$$

is added to the model whenever c_2-v_2 is a direct modification of c_1-v_1. Note that we allow the modified component to change its base name.

The transitive closure of $\mathit{delta}/4$ is expressed using the successor predicate $\mathit{succ}/4$:

$$\begin{aligned} \mathit{succ}(C_1, V_1, C_2, V_2) &\leftarrow \mathit{delta}(C_1, V_1, C_2, V_2) \\ \mathit{succ}(C_1, V_1, C_3, V_3) &\leftarrow \mathit{delta}(C_1, V_1, C_2, V_2), \\ &\qquad \mathit{succ}(C_2, V_2, C_3, V_3). \end{aligned} \tag{6}$$

We also define the predecessor predicate $\mathit{pred}/4$ analogously. A version v of c is current if it does not have any successors and the first if it does not have predecessors with the same base name:

$$\begin{aligned} \mathit{current\text{-}version}(C, V) \leftarrow& \\ &\mathit{component}(C, V), \\ &\text{not } \mathit{has\text{-}successor}(C, V) \end{aligned} \tag{7}$$

$$\begin{aligned} \mathit{first\text{-}version}(C, V) \leftarrow& \\ &\mathit{component}(C, V), \\ &\text{not } \mathit{has\text{-}predecessor}(C, V) \end{aligned} \tag{8}$$

where the auxiliary predicates are defined as follows:

$$\begin{aligned} \mathit{has\text{-}successor}(C, V_1) &\leftarrow \mathit{delta}(C, V_1, C, V_2) \\ \mathit{has\text{-}predecessor}(C, V_2) &\leftarrow \mathit{delta}(C, V_1, C, V_2) \end{aligned} \tag{9}$$

Branches and merges are modeled using the predicates $\mathit{branch}/3$, $\mathit{starts\text{-}branch}/2$, $\mathit{merge}/4$, and $\mathit{has\text{-}merge}/2$. Similarly to versions, each branch is identified by a constant. The fact:

$$\mathit{branch}(b, c, v) \leftarrow \tag{10}$$

is added to the model whenever the component c-v starts a new branch b. We identify those components that are first elements of some branches with $\mathit{starts\text{-}branch}/2$:

$$\mathit{starts\text{-}branch}(C, V) \leftarrow \mathit{branch}(B, C, V). \tag{11}$$

$\mathit{version\text{-}clause}(vc, c, op, v)$	$\mathit{any\text{-}clause}(vc, c)$
$\mathit{conjunction\text{-}clause}(vc)$	$\mathit{branch\text{-}clause}(vc, c, b)$
$\mathit{disjunction\text{-}clause}(vc)$	

Table 1: Predicates for repersenting version clauses

The fact

$$\mathit{merge}(b_1, b_2, c, v) \leftarrow \tag{12}$$

indicates that the branch b_1 is merged into the branch b_2 to produce the component c-v. Merging the branch b_1 into b_2 closes the branch b_1 but leaves b_2 open. If a two-way merge is desired, $\mathit{merge}/4$ should be true for both branches. The predicate $\mathit{has\text{-}merge}/2$ is used to indicate that a branch is eventually closed:

$$\mathit{has\text{-}merge}(B_1) \leftarrow \mathit{merge}(B_1, B_2, C, V). \tag{13}$$

4.3 Version Clauses

We divide the handling of version clauses into two parts. First, we want to find out what available components belong to the set that the clause represent. Second, we want to know whether the clause is satisfied in a configuration or not. For this, we define two predicates:

- $\mathit{in\text{-}clause}(c, v, vc)$ is true if c-v is in $V(vc)$; and
- $\mathit{satisfied}(vc)$ is true when vc is satisfied in the configuration.

We associate a unique identifier vc for each version clause. The version clauses are represented as facts that identify the type of the clause as well as relevant parameters. The type predicates are shown in Table 1.

We also have to store the u/l-cardinalities of clauses. For this we define the predicate $\mathit{cardinality}(vc, l, u)$.

Basic Clauses

We define $\mathit{in\text{-}clause}/3$ for basic clauses $(c\ op\ v)$ separately for all relational operators. For example, all components that are successors of the base component are compatible with the $\geq$-relation:

$$\begin{aligned}
&\mathit{in\text{-}clause}(C, V, VC) \leftarrow \\
&\qquad \mathit{version\text{-}clause}(VC, C, \geq, V), \\
&\qquad \mathit{available\text{-}component}(C, V) \\
&\mathit{in\text{-}clause}(C_2, V_2, VC) \leftarrow \\
&\qquad \mathit{version\text{-}clause}(VC, C_1, \geq, V_1), \\
&\qquad \mathit{available\text{-}component}(C_2, V_2), \\
&\qquad \mathit{succ}(C_1, V_1, C_2, V_2).
\end{aligned} \tag{14}$$

All other operators are defined in a similar manner. A basic clause is satisfied if the number of compatible components in a configuration is between the cardinality bounds l and u:

$$\begin{aligned}
\mathit{satisfied}(VC) \leftarrow L \leq \{&\mathit{in}(C', V') : \\
&\quad \mathit{in\text{-}clause}(C', V', VC)\} \leq U, \\
&\mathit{version\text{-}clause}(VC, C, \geq, V), \\
&\mathit{cardinality}(VC, L, U).
\end{aligned} \tag{15}$$

Branch Clauses

We define an auxiliary predicate $\mathit{in\text{-}branch}(b, c, v)$ to denote the fact that c-v belongs to the branch b. The branch point belongs always to b.

$$\mathit{in\text{-}branch}(B, C, V) \leftarrow \mathit{branch}(B, C, V). \tag{16}$$

We then recursively work up the delta-relation until we find the merge point or a point where a new branch has been created. The following rule captures the situation where the branch is open:

$$\begin{aligned}
\mathit{in\text{-}branch}(B, C, V_2) \leftarrow\ & \mathit{in\text{-}branch}(B, C, V_1), \\
& \mathit{succ}(C, V_1, C, V_2), \\
& \text{not } \mathit{starts\text{-}branch}(C, V2), \\
& \text{not } \mathit{has\text{-}merge}(C, B).
\end{aligned} \tag{17}$$

And similarily for closed branches:

$$\begin{aligned}
&\mathit{in\text{-}branch}(B_1, C, V_2) \leftarrow \\
&\qquad \mathit{in\text{-}branch}(B_1, C, V_1), \\
&\qquad \mathit{delta}(C, V_1, C, V_2), \\
&\qquad \text{not } \mathit{starts\text{-}branch}(C, V2), \\
&\qquad \mathit{merge}(C, B_1, B_2, V), \\
&\qquad V \neq V_2.
\end{aligned} \tag{18}$$

Having defined $\mathit{in\text{-}branch}/3$, it is easy to define the $\mathit{in\text{-}clause}/3$ rule for $(c\ \mathit{branch}\ b)$:

$$\begin{aligned}
\mathit{in\text{-}clause}(C, V, VC) \leftarrow\ & \mathit{branch\text{-}clause}(VC, C, B), \\
& \mathit{available\text{-}component}(C, V), \\
& \mathit{in\text{-}branch}(B, C, V).
\end{aligned} \tag{19}$$

The satisfaction of a branch clause is defined similarly to (15).

Any Clause

Any version of c is compatible with an any-clause.

$$\begin{aligned}
&\mathit{in\text{-}clause}(C, V, VC) \leftarrow \\
&\qquad \mathit{any\text{-}clause}(VC, C), \\
&\qquad \mathit{available\text{-}component}(C, V)
\end{aligned} \tag{20}$$

Again, the satisfaction is as by (15).

Connectives

The connectives are modeled using predicates $\mathit{conjunct}/2$ and $\mathit{disjunct}/2$. A conjunction $vc = vc_1 \wedge \cdots \wedge vc_n$ is expressed with n facts:

$$\mathit{conjunct}(vc, vc_1) \leftarrow \quad \cdots \quad \mathit{conjunct}(vc, vc_n) \leftarrow \tag{21}$$

A disjunction is defined in a similar manner. A component is compatible with a conjunction if it is compatible with all conjuncts:

$$\begin{aligned}
&\mathit{in\text{-}clause}(C, V, VC) \leftarrow \\
&\quad \mathit{in\text{-}clause}(C, V, VC') : \mathit{conjunct}(VC, VC'), \\
&\quad \mathit{conjunction\text{-}clause}(VC).
\end{aligned} \tag{22}$$

The satisfaction rule for $\mathit{conjunction\text{-}clause}/1$ is similar to rule (15). To see why we cannot define the satisfaction in terms of satisfaction of the conjuncts, consider the case of $((a \geq 1.0) \wedge (a \leq 1.5))$. Now the configuration $\{a\text{-}0.1, a\text{-}1.6\}$ satisfies both sub-clauses but the conjunction is not satisfied.

We do not define an $\mathit{in\text{-}clause}/3$ rule for disjunctions since we want to preserve the identities of the sets corresponding to the disjuncts. We may define the satisfaction of a disjunction directly in terms of its disjuncts. A disjunction with an u/l-cardinality is satisfied if the number of satisfied disjuncts is between l and u:

$$\begin{aligned} &\mathit{satisfied}(VC_1) \leftarrow \\ &\quad L \leq \{\mathit{satisfied}(VC_2) : \\ &\qquad \mathit{disjunct}(VC_1, VC_2)\} \leq U, \\ &\quad \mathit{disjunction\text{-}clause}(VC_1) \\ &\quad \mathit{cardinality}(VC_1, L, U). \end{aligned} \tag{23}$$

5 Creating Configurations

Our basic approach is tp divide the actual configuration process into two parts. First, we try to find what components have some reason to be in a configuration, and then we choose the configuration from the set of justified components.

5.1 Justification

A component may be in a configuration if we can somehow justify why it should be in there:

$$\{\mathit{in}(C_1, V_1)\} \leftarrow \mathit{justified}(C_1, V_1). \tag{24}$$

Intuitively, a component c is justified if

- it is a part of user requirements;
- another component in the configuration depends on it; or
- after including some component we have a choice to include c.

In the first two cases c absolutely has to be in the configuration but in the third case we may either put it in or leave it out.

When we have a dependency on or a choice over a version clause with an u/l-cardinality, we want to justify a corresponding number of components into the configuration. For example, consider the version clause $(A \geq 2.0)_0^2$ where the version space of A is as defined in Figure 2. Now the clause corresponds to the set:

$$V = \{A\text{-}2.0, A\text{-}3.0, A\text{-}3.2\}$$

Suppose that we had a choice over this set, with the $2/0$-cardinality. We can now include any subset of V that has at most two elements in the configuration so there are three possible ways to allocate justifications on V:

$$\begin{gathered} \{\mathit{justified}(A, 2.0), \mathit{justified}(A, 3.0)\} \\ \{\mathit{justified}(A, 2.0), \mathit{justified}(A, 3.2)\} \\ \{\mathit{justified}(A, 3.0), \mathit{justified}(A, 3.2)\} \end{gathered}$$

The reason why we cannot simply justify all components in V is that the choice allows us to include at most two components from V. If we justify all three components, we do not have any convenient way to prevent configurations where they all are in[3]. The next section presents the exact rules that are used to create justifications.

5.2 Dependency and Choice

We use the predicates $\mathit{depends}(c, v, vc)$ and $\mathit{choice}(c, v, vc)$ to denote that the corresponding relations hold between $c\text{-}v$ and the version clause vc.

We define the justifications in a two-stage process. In the first stage we keep track on what components justify others. This is done so that the correct number of justifications is created for each version clause. In the second stage we hide the originators of the justifications.

The first stage is implented by defining the following rule:

$$\begin{aligned} &L \leq \{\mathit{justifies}(C_1, V_1, C_2, V_2) : \\ &\quad \mathit{in\text{-}clause}(C_2, V_2, VC)\} \leq U \leftarrow \\ &\qquad \mathit{cardinality}(VC, L, U), \\ &\qquad \mathit{active\text{-}clause}(VC). \end{aligned} \tag{25}$$

Now $\mathit{justifies}(c_1, v_1, c_2, v_2)$ is true when the component $c_1\text{-}v_1$ gives us a reason to add $c_2\text{-}v_2$ to the configuration. The predicate $\mathit{active\text{-}clause}(vc)$ is an auxiliary that denotes that there is now some choice to make over the components in vc.

As the version clauses are in disjunctive normal form, the top-level clause A_l^u is always a disjunction. If we activate A, either because some component depends on or makes a choice over it, we activate some of the disjuncts of A:

$$\begin{aligned} &L \leq \{\mathit{active\text{-}clause}(VC_2) : \\ &\quad \mathit{disjunct}(VC_1, VC_2)\} \leq U \leftarrow \\ &\qquad \mathit{disjunctive\text{-}clause}(VC_1), \\ &\qquad \mathit{cardinality}(VC_1, L, U), \\ &\qquad \mathit{depends}(C, V, VC_1), \\ &\qquad \mathit{in}(C, V). \end{aligned} \tag{26}$$

An identical rule is defined also for $\mathit{choice}/3$.

The second stage of the justification creation can be implemented using only single rule:

$$\mathit{justified}(C_2, V_2) \leftarrow \mathit{justifies}(C_1, V_1, C_2, V_2) \tag{27}$$

since a component is justified if any component justifies it.

Finally, we assert that a dependency has always be satisfied:

$$\begin{aligned} \leftarrow\ &\text{not } \mathit{satisfied}(VC), \\ &\mathit{depends}(C, V, VC), \\ &\mathit{in}(C, V). \end{aligned} \tag{28}$$

5.3 Incompatibility

The incompatibility relation is the simplest one to model since it does not create justifications. We us the predicate $\mathit{conflicts}(c, v, vc)$ to denote that it is an error if $c\text{-}v$ is in a configuration when vc is also satisfied:

$$\begin{aligned} \leftarrow\ &\mathit{satisfied}(VC), \\ &\mathit{conflicts}(C, V, VC), \\ &\mathit{in}(C, V). \end{aligned} \tag{29}$$

[3]That is, if the version clause corresponds to the choice relation. With the dependency relation we demand that the clause also has to be satisfied so this problem does not exist there.

5.4 Example

Consider again the situation in Figure 2. Suppose that A-3.2 has to be in the configuration and that it depends on the version clause C_2 of Example 2.1. Let us denote the subclauses of C_2 by $C_{21} = (b > 0.7)$ and $C_{22} = (c < 1.1)$.

By rule (26) we deduce that exactly one of

$$\{active\text{-}clause(C_{21}), active\text{-}clause(C_{22})\}$$

has to be true. If the first one is true, then by the rule (25) we know that B-0.8 has to be justified, so we deduce $justifies(A, 3.2, B, 0.8)$. Next, using the rule (27) we deduce $justified(B, 0.8)$. Now, by (24) we may either include B-0.8 in the configuration or not. However, the rule (28) forces us to include it and we get $in(B, 0.8)$. Nothing else can be added so the first configuration $\mathcal{C}_1 = \{A\text{-}3.2, B\text{-}0.8\}$.

In the second case we choose to activate C_{22}. Using a similar deduction we come into the conclusion that $in(C\text{-}1.0)$ has to be true in the answer set so C-1.0 is in the configuration along with A-3.2 and $\mathcal{C}_2 = \{A\text{-}3.2, C\text{-}1.0\}$.

5.5 Abstract Components

Sometimes we do not want to specify the components explicitly while using version clauses. For example, we may want to say that a component A depends on functionality F but we are not interested in what particular component provides it. The most simple approach to handle these cases is to define an abstract component f and make it depend on the components that provide the functionality. A similar approach is used in the current Debian GNU/Linux system to define collections of components [Debian, 2001].

6 Implementation

We constructed a test implementation of the configuration model using the SMODELS logic programming system [Simons, 2000]. The model works with the **smodels** version 2.26 and **lparse** version 1.0.3. The model has not been used in any practical applications, yet, but we are planning to add it to our formalization of the Debian GNU/Linux system [Syrjänen, 1999; 2000a; 2000b].

7 Conclusions

We presented a method for representing software configuration version spaces using logic programs with the stable model semantics. We defined a simple language for expressing version relationships in terms of version clauses. A version clause denotes a set of versions of a component and they may be combined using conjunction and disjunction connectives. However, for implementation-specific reasons the clauses have to be expressed in disjunctive normal form.

We formalized the version clauses into extended high-level logic programming rules and formalized the three basic configuration constraints, dependency, incompatibility, and choice, using them. The dependency and choice constraints do not directly add components into a configuration but do that via justfications.

We have produced only a prototype implementation of the model and have not tested it on any realistic system. In future, we are planning to test it by incorporating it into our model of the Debian GNU/Linux system.

Acknowledgements

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

References

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

[Debian, 2001] Debian GNU/Linux, 2001. Available at: <url:http://www.debian.org>.

[Gelfond and Lifschitz, 1988] 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.

[Niemelä and Simons, 2000] Ilkka Niemelä and Patrik Simons. Extending the smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer Academic Publishers, 2000.

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

[Simons, 2000] P. Simons. Extending and implementing the stable model semantics. Research Report 58, Helsinki University of Technology, Helsinki, Finland, 2000.

[Soininen and Niemelä, 1999] 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.

[Soininen *et al.*, 2001] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge*, Stanford, USA, March 2001.

[Syrjänen, 1999] Tommi 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.

[Syrjänen, 2000a] Tommi Syrjänen. Including diagnostic information in configuration models. In *Proceedings of the First International Conference on Computational Logic*, pages 837–851, London, UK, July 2000. Springer-Verlag.

[Syrjänen, 2000b] Tommi Syrjänen. Optimizing configurations. In *Proceedings of the ECAI Workshop W02 on Configuration*, pages 85–90, Berlin, Germany, August 2000.