Developing a Declarative Rule Language for Applications in Product Configuration

Timo Soininen[1] and Ilkka Niemelä[2]

[1] Helsinki University of Technology, TAI Research Center and Lab. of Information Processing Science, P.O.Box 9555, FIN-02015 HUT, Finland
Timo.Soininen@hut.fi

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

Abstract. A rule-based language is proposed for product configuration applications. It is equipped with a declarative semantics providing formal definitions for main concepts in product configuration, including configuration models, requirements and valid configurations. The semantics uses Horn clause derivability to guarantee that each element in a configuration has a justification. This leads to favorable computational properties. For example, the validity of a configuration can be decided in linear time and other computational tasks remain in **NP**. It is shown that CSP and dynamic CSP can be embedded in the proposed language which seems to be more suitable for representing configuration knowledge. The rule language is closely related to normal logic programs with the stable model semantics. This connection is exploited in the first implementation which is based on a translator from rules to normal programs and on an existing high performance implementation of the stable model semantics, the Smodels system.

1 Introduction

Product configuration has been a fruitful topic of research in artificial intelligence for the past two decades (see, e.g. [10, 15, 1, 8]). In the last five years product configuration has also become a commercially successful application of artificial intelligence techniques. Knowledge-based systems (KBS) employing techniques such as constraint satisfaction (CSP) [19] have been applied to product configuration. However, the product configuration problem exhibits dynamic aspects which are difficult to capture in, e.g., the CSP formalism. The choices form chains where previous choices affect the set of further choices that need to be made. In addition, making a choice needs to be justified by a chain of previous choices. This has led to the development of extensions of the CSP formalism, such as dynamic constraint satisfaction (DCSP) [11] and generative constraint satisfaction (GCSP) [7].

In this paper, which is a revised version of [17], we present work-in-progress on developing a logic programming like rule language for product configuration applications. The rule language is defined with the goal that relevant knowledge in the configuration domain can be represented compactly and conveniently. We provide a simple declarative semantics for the language which guarantees a justification for each choice.

We study the complexity of the relevant computational tasks for this language. The main result is that the task of finding a configuration is **NP**-complete and that the validity of a configuration can be checked in linear time. We also show that our language can be seen as a generalization of the CSP and DCSP formalisms. There are local and linear solution preserving mappings from the CSP and DCSP formalisms to the language, but mapping in the other direction is difficult. This is due to the difficulty of capturing justifications in CSP and to more expressive rules that seem difficult to capture in DCSP.

The semantics of the rule language is closely related to the declarative semantics of logic programs. This relation is exploited in developing the first implementation of the language. We present a solution preserving local and polynomial translation from the rule language to normal logic programs with the stable model semantics [6]. Our implementation is based on an existing high performance implementation of the stable model semantics for normal logic programs, the Smodels system [12, 13]. For the implementation it is enough to build a front-end to the Smodels system realizing the translation to normal programs. In order to estimate the feasibility of our approach we study two simple configuration problems. We observe that such examples are straightforward to model in our language and that our implementation exhibits reasonable performance.

2 Product Configuration Domain

Product configuration is roughly defined as the problem of producing a specification of a product individual as a collection of predefined components. The inputs of the problem are a *configuration model*, which describes the components that can be included in the configuration and the rules on how they can be combined to form a working product, and *requirements* that specify some properties that the product individual should have. The output is a *configuration*, an accurate enough description of a product individual to be manufactured. The configuration must *satisfy* the requirements and be *valid* in the sense that it does not break any of the rules in the configuration model and it consists only of the components that have justifications in terms of the configuration model.

This definition of product configuration does not adequately capture all aspects of configuration problems. Missing features include representing and reasoning about attributes, structure and connections of components, resource production and use by components [15, 1, 7] and optimality of a configuration. Our definition is a simplification that nonetheless contains the core aspects of configuration problem solving. It is intended as the foundation on which further aspects of product configuration can be defined. Correspondingly, we use the term *element* to mean any relevant piece of information on a configuration. An element can be a component or information on, e.g., the structure of a product.

A *product configurator* is a KBS that is capable of representing the knowledge included in configuration models, requirements and configurations. In addition, it is capable of (i) *checking* whether a configuration is valid with respect to the configuration model and satisfies a set of requirements and/or (ii) *generating* one or all valid configuration(s) for a configuration model and a set of requirements.

Example 1. As an example of a configurable product, consider a PC. The components in a typical configuration model of a PC include different types of display units, hard disks, CD ROM drives, floppy drives, extension cards and so on. These have rules on how they can be combined with each other to form a working product. For example, a PC would typically be defined to have a mass storage which must be chosen from a set of alternatives, e.g. an IDE hard disk, SCSI hard disk and a floppy drive. A computer would also need a keyboard, which could have either a Finnish or United Kingdoms layout. Having a SCSI hard disk in the configuration of a PC would typically require that an additional SCSI controller is included in the configuration as well. In addition, a PC may optionally have a CD ROM drive. A configuration model for a PC might also define that unless otherwise specified, an IDE hard disk will be the default choice for mass storage.

The fundamental form of knowledge in a configuration model is that of a *choice* [18]. There are basically two types of choices. Either at least one or exactly one of alternative elements must be chosen. Whether a choice must be made may depend on some set of elements. Other forms of configuration knowledge include the following:

- A set of elements in the configuration *requires* some set of elements to be in the configuration as well [18, 8].
- A set of elements are *incompatible* with each other [18, 8].
- An element is *optional*. Optional elements can be chosen into a configuration or they can be left out.
- An element is a *default*. It is in the configuration unless otherwise specified.

3 Configuration Rule Language

In this section we define a configuration rule language **CRL** for representing configuration knowledge. The idea is to focus on interactions of the elements and not on details of a particular configuration knowledge modeling language. For simplicity, we have kept the number of primitives in the language low by focusing on choices and requires and incompatibility interactions. Extending the language with optional and default choices is straightforward (see Example 4).

The basic construction blocks of the language are propositional atoms, which are combined through a set of connectives into rules. We assume for simplicity that atoms can be used to represent elements adequately. We define a *configuration model* and *requirements* as sets of **CRL** rules. A *configuration* is defined as a set of atoms.

The syntax of **CRL** is defined as follows. The alphabet of **CRL** consists of the connectives ",", "$\leftarrow$", "$|$", "$\oplus$", "not", parentheses and atomic propositions. The connectives are read as "and", "requires", "or", "exclusive or" and "not", respectively. The rules in **CRL** are of the form

$$a_1\theta\cdots\theta a_l \leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n)$$

where $\theta \in \{|,\oplus\}$, $a_1,\ldots,a_l$, $b_1,\ldots,b_m$, $c_1,\ldots,c_n$ are atoms and $l \geq 0$, $m \geq 0$, $n \geq 0$. We refer to the subset of a set of rules R with exactly one atom in the head as *requires-rules*, R_r, rules with more than one atom in the head separated by "$|$" as *choice-rules*, rules with more than one atom in the head separated by "$\oplus$" as *exclusive choice-rules* R_e, and rules with no atoms in the head as *incompatibility-rules*, R_i. In the definitions below we treat requires-rules as a special case of choice-rules with only one alternative in the head.

Example 2. A very simple configuration model R_{PC} of the PC in Example 1 (without the optional CD-ROM and default mass storage) could consist of the following rules:

$$\begin{array}{l}
\mathit{computer} \leftarrow \\
\mathit{IDEdisk} \mid \mathit{SCSIdisk} \mid \mathit{floppydrive} \leftarrow \mathit{computer} \\
\mathit{FinnishlayoutKB} \oplus \mathit{UKlayoutKB} \leftarrow \mathit{computer} \\
\mathit{SCSIcontroller} \leftarrow \mathit{SCSIdisk}
\end{array}$$

Next we define when a configuration satisfies a set of rules and is valid with respect to a set of rules. We say that a configuration *satisfies requirements* if it satisfies the corresponding set of rules.

Definition 1. *A configuration C* satisfies *a set of rules R in* **CRL**, *denoted by $C \models R$, iff the following conditions hold:*

(i) If $a_1 \mid \cdots \mid a_l \leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n) \in R_\mathrm{r} \cup R_\mathrm{c}$, $\{b_1,\ldots,b_m\} \subseteq C$, and $\{c_1,\ldots c_n\} \cap C = \emptyset$, then $\{a_1,\ldots,a_l\} \cap C \neq \emptyset$.

(ii) If $a_1 \oplus\cdots\oplus a_l \leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n) \in R_\mathrm{e}$, $\{b_1,\ldots,b_m\} \subseteq C$, and $\{c_1,\ldots c_n\} \cap C = \emptyset$, then for exactly one $a \in \{a_1,\ldots,a_l\}$, $a \in C$.

(iii) If $\leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n) \in R_i$, then it is not the case that $\{b_1,\ldots,b_m\} \subseteq C$ and $\{c_1,\ldots c_n\} \cap C = \emptyset$ hold.

In order to define the validity of a configuration, we employ an operator R^C that is a transformation of a set of rules R in **CRL**.

Definition 2. *Given a configuration C and a set of rules R in* **CRL**, *we denote by R^C the set of rules*

$$\{a_i \leftarrow b_1,\ldots,b_m : a_1\theta\cdots\theta a_l \leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n) \in R, \theta \in \{|,\oplus\},$$
$$a_i \in C, 1 \leq i \leq l, \{c_1,\ldots c_n\} \cap C = \emptyset\}$$

The result of the transformation is a set of Horn clauses if we interpret the symbols "$\leftarrow$" and "," as classical implication and conjunction, respectively. Under this interpretation the reduct R^C has a unique least model, which we denote by

$\mathrm{MM}(R^C)$. Notice that the least model of a set of Horn clauses coincides with the set of atoms logically entailed by them and also with the set of atoms derivable by interpreting them as inference rules. The intuition behind the transformation is that, given a choice-rule, if any of the alternatives in the head of the rule are chosen, then the reduct of the transformation includes a rule that can justify the choice (if the body of the rule can be justified). If some alternative is not chosen, then there is no need for the choice to be justified and consequently no corresponding rules are included. The default negation "not(·)" is handled using a technique similar to that in the stable model semantics of logic programs [6].

Definition 3. *Given a configuration C and a set of rules R in* **CRL**, *C is R-valid iff $C = \mathrm{MM}(R^C)$ and $C \models R$.*

The idea of the definition is as follows: the first fix-point condition guarantees that a configuration must be justified by the rules. All the things in the configuration are derivable from (the reduct of) the configuration rules. On the other hand, everything that can be derived using (the reduct of) the rules must be in the configuration. The second condition ensures that all the necessary choices have been made and all the requires and incompatibility-rules are respected.

Example 3. Consider the configuration model R_{PC} in Example 2, the simple set of requirements $\{FinnishlayoutKB \leftarrow\}$ and the configurations

$$\begin{aligned} C_1 &= \{computer, SCSIdisk, UKlayoutKB\} \\ C_2 &= \{computer, IDEdisk, FinnishlayoutKB, SCSIcontroller\} \\ C_3 &= \{computer, SCSIdisk, FinnishlayoutKB, SCSIcontroller\} \end{aligned}$$

The configuration C_1 does not satisfy the configuration model nor the requirements according to Definition 1 and thus it is not R_{PC}-valid, either. The configuration C_2 does satisfy the configuration model and the requirements. However, it is not R_{PC}-valid because the reduct ${R_{PC}}^{C_2}$ is

$$\{computer \leftarrow; IDEdisk \leftarrow computer; FinnishlayoutKB \leftarrow computer;$$
$$SCSIcontroller \leftarrow SCSIdisk\}$$

The minimal model $\mathrm{MM}({R_{PC}}^{C_2}) = \{computer, IDEdisk, FinnishlayoutKB\}$ does not contain $SCSIcontroller$ and thus it is not equal to C_2. The configuration C_3 is R_{PC}-valid and satisfies the requirements.

Example 4. Consider the following sets of rules:

$R_1:$	$R_2:$	$R_3:$
$a \mid b \leftarrow c$	$a \mid b \leftarrow c$	$a \mid b \leftarrow c$
$c \leftarrow$	$c \oplus c' \leftarrow d$	$c \oplus c' \leftarrow d$
	$d \leftarrow$	$a \leftarrow \mathrm{not}(b), d$
		$d \leftarrow$

The valid configurations with respect to R_1 are $\{c, a\}$, $\{c, b\}$ and $\{c, a, b\}$. The reducts of R_1 with respect to these configurations are $\{a \leftarrow c; c \leftarrow\}$, $\{b \leftarrow c; c \leftarrow\}$

and $\{a \leftarrow c; b \leftarrow c; c \leftarrow\}$, respectively. Clearly, the minimal models of these reducts coincide with the configurations and the configurations satisfy the rules in R_1. On the other hand, if the latter rule is omitted, the only valid configuration is the empty configuration $\{\}$, since a and b cannot have a justification.

Although **CRL** does not include primitives for some typical forms of configuration knowledge such as *optional choices* and *default* alternatives, they can be captured fairly straightforwardly. The first two rules in R_2 demonstrate how to represent an optional choice-rule whose head consists of the atoms a and b and whose body is d. The valid configurations with respect to R_2 are $\{c', d\}$, $\{a, c, d\}$, $\{b, c, d\}$ and $\{a, b, c, d\}$. In this example either c or c' must be in a configuration. These additional atoms represent the cases where the choice is made and not made, respectively. Now, consider the rule set R_3 obtained by adding the rule $a \leftarrow \mathrm{not}(b), d$ to R_2. The valid configurations are now $\{c', a, d\}$, $\{c, a, d\}$, $\{c, b, d\}$ and $\{c, a, b, d\}$. This rule set represents a default choice (a is the default) which is made unless one of the alternatives is explicitly chosen.

4 Relationship to Logic Programming Semantics

The configuration rule language **CRL** resembles disjunctive logic programs and deductive databases. The main syntactic difference is that two disjunctive operators are provided whereas in disjunctive logic programming typically only one is offered. The semantics is also similar to logic programming semantics. The main difference is that leading disjunctive semantics (see, e.g., [3, 5]) have minimality of models as a built-in property whereas our semantics does not imply subset minimality of configurations. The rule set R_1 above is an example of this. However, there are semantics allowing non-minimal models and, in fact, if we consider the subclass with one disjunctive operator, i.e. ordinary choice-rules, our notion of a valid configuration coincides with possible models introduced by Sakama and Inoue [14] for disjunctive programs. They observed that possible models of disjunctive programs can be captured with stable models of normal programs by a suitable translation of disjunctive programs to non-disjunctive programs [14]. Here we extend this idea to exclusive choice-rules and present a slightly different, more compact and computationally oriented translation.

Given a set of rules R in **CRL** the corresponding normal logic program is constructed as follows. The requires-rules R_{r} are taken as such. The incompatibility-rules R_{i} are mapped to logic program rules with the same body but a head f and a new rule $f' \leftarrow \mathrm{not}(f'), f$ is included where f, f' are new atoms not appearing in R. For each choice-rule

$$a_1 \mid \cdots \mid a_l \leftarrow b_1, \ldots, b_m, \mathrm{not}(c_1), \ldots, \mathrm{not}(c_n)$$

in R_{c} we include a rule $f \leftarrow b_1, \ldots, b_m, \mathrm{not}(c_1), \ldots, \mathrm{not}(c_n), \hat{a}_1, \ldots, \hat{a}_l$ and for all $i = 1, \ldots, l$, two rules

$$a_i \leftarrow \mathrm{not}(\hat{a}_i), b_1, \ldots, b_m, \mathrm{not}(c_1), \ldots, \mathrm{not}(c_n) \quad \text{and} \quad \hat{a}_i \leftarrow \mathrm{not}(a_i)$$

where $\hat{a_1}, \ldots, \hat{a_l}$ are new atoms. Each exclusive choice-rule is translated the same way as an ordinary choice-rule except that we include additionally the set of rules of the form $f \leftarrow b_1, \ldots, b_m, \mathrm{not}(c_1), \ldots, \mathrm{not}(c_n), a', a''$ where $a' = a_i, a'' = a_j$ for some $i, j, 1 \leq i < j \leq l$. Note that the number of the additional rules is quadratic in the number of head atoms, but for ordinary choice-rules the translation is linear. Now the stable models of the program provide the valid configurations for the rules. The close correspondence implies that an implementation of the stable model semantics can be used for configuration tasks.

5 Complexity Issues

In this section we briefly consider the complexity of the following key decision problems in configuration: (i) **C-SAT**: decide whether a configuration satisfies a set of rules, (ii) **EXISTS**: determine whether there is a valid configuration for a set of rules, and (iii) **QUERY**: decide whether there is a valid configuration C for a set of rules satisfying a set of requirements Q ($C \models Q$).

First, we observe that **C-SAT** is decidable in linear time. Second, we note that checking whether a set of atoms is a valid configuration can be done in linear time. This holds as for a set of rules and a candidate configuration, the reduct can be computed in linear time and, similarly, the unique least model of a set of Horn clauses is computable in linear time [4]. This implies that the major computational tasks in configuration using our semantics are in **NP**.

For **EXISTS** and **QUERY**, we consider some subclasses of **CRL** to show the boundary for **NP**-completeness. For example, $\mathbf{CRL_r}$ is the subset where only requires-rules are allowed, $\mathbf{CRL_{rd}}$ permits additionally default negations, $\mathbf{CRL_{re}}$ allows exclusive choice-rules in addition to requires-rules and $\mathbf{CRL_{rci}}$ admits requires-rules, choice-rules and incompatibility-rules. The results are summarized in Table 1. They are fairly straightforward to demonstrate (see [17] for more details). Most of the results can also be established from the complexity results for the possible model semantics [14, 5].

Table 1. Complexity results for configuration tasks

Language	**C-SAT**	**EXISTS**	**QUERY**
$\mathbf{CRL_r}$	Poly	Poly	Poly
$\mathbf{CRL_{ri}}$	Poly	Poly	Poly
$\mathbf{CRL_{rc}}$	Poly	Poly	**NP**-compl.
$\mathbf{CRL_{rd}}$	Poly	**NP**-compl.	**NP**-compl.
$\mathbf{CRL_{re}}$	Poly	**NP**-compl.	**NP**-compl.
$\mathbf{CRL_{rci}}$	Poly	**NP**-compl.	**NP**-compl.

6 Relation to Constraint Satisfaction

Configuration is often cast as a constraint satisfaction or dynamic constraint satisfaction problem. In this section we aim to show that **CRL** contains CSP and DCSP as special cases and is an extension of these two approaches. We note that for all the formalisms dealt with in this section the problem corresponding to generating a configuration is **NP**-complete.

6.1 Mapping Constraint Formalisms to CRL

We first recall that a CSP consists of a set of *variables*, a set of possible *values* for each variable, called the *domain* of the variable, and a set of *constraints*. We assume in the following that the domains are finite. A constraint defines the allowed combinations of values for a set of variables by specifying a subset of the Cartesian product of the domains of the variables. A *solution* to a CSP is an *assignment* of values to all variables such that the constraints are satisfied, i.e., the value combinations are allowed by at least one tuple of each constraint.

A DCSP is an extension of a CSP that also has of a set of variables, domains, and constraints (called here *compatibility constraints*). However, all the variables need not be given a value, i.e., be *active* in a solution. A DCSP additionally defines a set of *initial variables* that must be active in every solution and a set of *activity constraints*. An activity constraint states either that if a given condition is true then a certain variable is active, or that if a given condition is true, then a certain variable must not be active. The condition may be expressed as a compatibility constraint (*require* and *require not* activity constraints) or it may state that some other variable is active (*always require* and *always require not* activity constraints). A solution to a DCSP is an assignment of values to variables such that it (i) fulfills the compatibility and activity constraints, (ii) contains assignments for the initial variables, and (iii) is minimal.

We next define a mapping from the DCSP formalism to **CRL**. We note that as CSP is a special case of DCSP with no activity constraints and with all variables in the set of initial variables, the same mapping can be used for a CSP. In the mapping from a DCSP to **CRL** representation we introduce (i) a new distinct atom for each variable, v_i, to encode its activity, (ii) a new distinct atom $sat(c_i)$ for each compatibility constraint c_i, and (iii) a new distinct atom $v_i(val_{i,j})$ for each variable v_i and value $val_{i,j}$ in the domain of v_i.

Each initially active variable v_i is mapped to a fact $v_i \leftarrow$. Each variable v_i and its domain $\{val_{i,1}, \ldots, val_{i,n}\}$ is mapped to an exclusive choice-rule of the following form: $v_i(val_{i,1}) \oplus \cdots \oplus v_i(val_{i,n}) \leftarrow v_i$. A compatibility constraint on variables $v_1, \ldots, v_n$ is represented using a set of requires-rules of form $sat(c_i) \leftarrow v_1(val_{1,j}), v_2(val_{2,k}), \cdots, v_n(val_{n,l})$, one rule for each allowed value combination $val_{1,j}, \ldots, val_{n,l}$. An incompatibility-rule of the form $\leftarrow v_1, \ldots, v_n, \mathrm{not}(sat(c_i))$ is included to enforce the constraint.

Example 5. Given a CSP with two variables, $package$ and $frame$ with domains $\{luxury, deluxe, standard\}$ and $\{convertible, sedan, hatchback\}$, respectively, and a constraint $c_1 = \{\{luxury, convertible\}, \{standard, hatchback\}\}$ on *package* and *frame*, the following rule set is produced by the mapping:

$$
\begin{aligned}
&package \leftarrow \\
&frame \leftarrow \\
&package(luxury) \oplus package(deluxe) \oplus package(standard) \leftarrow package \\
&frame(convertible) \oplus frame(sedan) \oplus frame(hatchback) \leftarrow frame \\
&sat(c_1) \leftarrow package(luxury), frame(convertible) \\
&sat(c_1) \leftarrow package(standard), frame(hatchback) \\
&\leftarrow package, frame, \mathrm{not}(sat(c_1))
\end{aligned}
$$

An always require activity constraint is mapped to a requires-rule $v_2 \leftarrow v_1$ where v_2 is the activated variable and v_1 is the condition variable. An always require not activity constraint is mapped to an incompatibility-rule $\leftarrow v_1, v_2$ where v_1 and v_2 are the condition and deactivated variables, respectively. A require variable activity constraint is mapped to a set of requires-rules, one rule of the form $u \leftarrow v_1(val_{1,j}), \ldots, v_n(val_{n,k})$ for each allowed value combination $\{val_{1,j}, \ldots, val_{n,k}\}$ of variables $v_1, \ldots, v_n$, where u is the activated variable. A require not activity constraint is mapped to a set of incompatibility-rules, one rule of the form $\leftarrow u, v_1(val_{1,j}), \ldots, v_n(val_{n,k})$ for each allowed value combination $\{val_{1,j}, \ldots, val_{n,k}\}$ of variables $v_1, \ldots, v_n$ where u is the deactivated variable.

Example 6. Given a DCSP with two variables, *package* and *sunroof*, whose domains are $\{luxury, deluxe, standard\}$ and $\{sr_1, sr_2\}$, respectively, a set of initial variables $\{package\}$ and a require activity constraint that if *package* has value *luxury*, then *sunroof* is active, the following rule set is produced:

$$
\begin{aligned}
&package \leftarrow \\
&package(luxury) \oplus package(deluxe) \oplus package(standard) \leftarrow package \\
&sunroof(sr_1) \oplus sunroof(sr_2) \leftarrow sunroof \\
&sunroof \leftarrow package(luxury)
\end{aligned}
$$

It is easy to see that each valid configuration is a solution to the DCSP and vice versa. The minimality of solutions can be shown by noting that the rules that can cause a variable to be active can be translated to normal logic programs. For this subclass of rules the configurations coincide with stable models which are subset minimal [6]. The size of the resulting rule set is linear in the size of the DCSP problem instance. The mapping is *local* in the sense that each variable and its domain, initial variable, compatibility constraint and activity constraint can be mapped separately from the other elements of the problem instance.

6.2 Expressiveness of CRL vs. CSP

Next we argue that **CRL** is strictly more expressive than CSP by using the concept of *modularity.* A modular representation in some formalism is such that a small, local change in the knowledge results in a small change in the representation. This property is important for easy maintenance of a knowledge base.

We show that under mild assumptions the CSP formalism cannot modularly capture the justifications of a configuration. We say that **CRL** is *modularly representable* by CSP iff for every set of **CRL** rules there is a CSP such that rules are represented in the CSP independent of the representation of the basic facts (i.e. requires-rules with empty bodies) so that a change in the facts does not lead to a change involving both additions and removals of either allowed tuples, constraints, variables or values. In addition, the solutions to the CSP must *agree* with the **CRL** configurations in that (i) the truth values of the atoms in a configuration can be read from the values of Boolean CSP variables representing the atoms and (ii) these variables have the same truth values as the corresponding atoms.

Theorem 1. CRL *is not modularly representable by CSP.*

Proof. Consider the set of rules $R = \{c \leftarrow b\}$ and assume that it can be modularly represented by a CSP. Hence, there is a CSP $\mathrm{T}(R)$ such that in all the solutions of $\mathrm{T}(R)$ the variables representing atoms b and c in the configuration language have the value $false$ as R has the empty set as its unique valid configuration. Consider now a set of facts $F = \{b \leftarrow\}$. The configuration model $R \cup F$ has a unique valid configuration $\{b, c\}$. This means that $\mathrm{T}(R)$ updated with F must not have a solution in which variables encoding b and c have the value $false$. In addition, $\mathrm{T}(R)$ updated with F must have at least one solution in which the atoms encoding b and c have the value $true$. It can be shown that changes including either only additions or only removals of either allowed tuples, constraints, variables or values cannot both add solutions and remove them, which is a contradiction and hence the assumption is false.

The fact that there is no modular representation of **CRL** in the CSP formalism is caused by the justification property of **CRL** which introduces a non-monotonic behavior. A similar argument can therefore be used for showing a similar result for, e.g., propositional logic [17]. We note that the question whether there is a modular representation of a configuration model given in **CRL** as a DCSP is open. The DCSP formalism exhibits a non-monotonic behavior, so a similar argument cannot be used for this case. It can be used, however, to show that there is no modular representation of a DCSP as a CSP. Representing **CRL** as DCSP does not seem straightforward, as the DCSP approach does not directly allow activity constraints that have a choice among a set of variables to activate or default negation in the condition part.

7 Implementation

In this section we describe briefly our implementation of **CRL**, demonstrate the use of **CRL** with a car configuration problem from [11] and provide information on performance of the implementation for the car problem.

Our implementation of **CRL** is based on the translation of **CRL** to normal logic programs presented in Sect. 4 and on an existing high performance implementation of the stable model semantics, the Smodels system [12, 13]. This system seems to be the most efficient implementation of the stable model semantics currently available. It is capable of handling large programs, i.e. over 100 000 ground rules, and has been applied successfully in a number of areas including planning [2], model checking for distributed systems [9], and propositional satisfiability checking [16].

We have built a front-end to Smodels which takes as input a slightly modified (see below) set of **CRL** rules and transforms it to a normal logic program whose stable models correspond to valid configurations. Then Smodels is employed for generating stable models. The implementation can generate a given number of configurations, all of them, or the configurations that satisfy requirements given as a set of literals.

Smodels is publicly available at `http://www.tcs.hut.fi/pub/smodels/`. The front-end is included in the new parser of Smodels, `lparse`, which accepts in addition to normal program rules (requires-rules) also "inclusive" choice-rules and incompatibility-rules. Exclusive choice-rules are supported by rules of the form $\leftarrow n\{a_1, \ldots, a_l\}$ where n is an integer. The rule acts like an integrity constraint eliminating models, i.e. configurations, with n or more of the atoms from $\{a_1, \ldots, a_l\}$. This allows a succinct coding of, e.g., exclusiveness without the quadratic overhead which results when using normal rules. Hence, an exclusive choice-rule $a_1 \oplus \cdots \oplus a_l \leftarrow Body$ can be expressed as a combination of an "inclusive" choice-rule $a_1 \mid \cdots \mid a_l \leftarrow Body$ and the rule $\leftarrow Body, 2\{a_1, \ldots, a_l\}$.

Our first example, **CAR**, was originally defined as a DCSP [11]. In Fig. 1 the problem is translated to **CRL** using the mappings defined in the previous section with the exception that the compatibility constraints are given a simple rule form similar to that in [11]. There are several choices of packages, frames, engines, batteries and so on for a car. At least a package (*pack*), frame and engine must be chosen from the alternatives specified for them. Choosing a particular alternative in a choice-rule can make other choices necessary. For example, if the package is chosen to be luxury (l), then a sunroof and an airconditioner (*aircond*) must be chosen as well. In addition, some combinations of alternatives are mutually exclusive, e.g., the luxury alternative for package cannot be chosen with the $ac1$ alternative for airconditioner. The second example, **CARx2**, is modified from **CAR** by doubling the size of the domain of each variable. In addition, for each new value and each compatibility and activity constraint in the original example a new similar constraint referring to the new value is added.

We did some experiments with the two problems in **CRL** form. The tests were run on a Pentium II 233 MHz with 128MB of memory, Linux 2.0.35 operating system, `smodels` version 1.12 and `lparse` version 0.9.19. The test cases are available at `http://www.tcs.hut.fi/pub/smodels/tests/padl99.tar.gz`. Table 2 presents the timing results for computing one and all valid configurations, the number of valid configurations found and the size of the initial search space which is calculated by multiplying the number of alternatives for each choice. The execution times include reading and parsing the set of input rules, its translation to a normal program as well as outputting the configurations in a file. The times were measured using the Unix `time` command and they are the sum of

$pack(l) \oplus pack(dl) \oplus pack(std) \leftarrow pack$
$frame(conv) \oplus frame(sedan) \oplus frame(hb) \leftarrow frame$
$engine(s) \oplus engine(m) \oplus engine(l) \leftarrow engine$
$battery(s) \oplus battery(m) \oplus battery(l) \leftarrow battery$
$sunroof(sr1) \oplus sunroof(sr2) \leftarrow sunroof$
$aircond(ac1) \oplus aircond(ac2) \leftarrow aircond$
$glass(tinted) \oplus glass(nottinted) \leftarrow glass$
$opener(auto) \oplus opener(manual) \leftarrow opener$
$battery(m) \leftarrow opener(auto), aircond(ac1)$
$battery(l) \leftarrow opener(auto), aircond(ac2)$
$\leftarrow sunroof(sr1), aircond(ac2), glass(tinted)$
$\leftarrow pack(std), aircond(ac2)$
$\leftarrow pack(l), aircond(ac1)$
$\leftarrow pack(std), frame(conv)$

$pack \leftarrow$
$frame \leftarrow$
$engine \leftarrow$
$sunroof \leftarrow pack(l)$
$aircond \leftarrow pack(l)$
$sunroof \leftarrow pack(dl)$
$opener \leftarrow sunroof(sr2)$
$aircond \leftarrow sunroof(sr1)$
$glass \leftarrow sunroof$
$battery \leftarrow engine$
$sunroof \leftarrow opener$
$sunroof \leftarrow glass$
$\leftarrow sunroof(sr1), opener$
$\leftarrow frame(conv), sunroof$
$\leftarrow battery(s), engine(s),$
$aircond$

Fig. 1. Car configuration example

user and system time. The test results show that for this small problem instance the computation times are acceptable for interactive applications. For example, in the larger test case it takes on average less than 0.0004 s to generate a configuration. We are not aware of any other reported test results for solving this problem in the DCSP or any other form.

Table 2. Results from the car example

Problem	Initial search space	Valid configurations	one	all
CAR	1 296	198	0.06 s	0.15 s
CARx2	331 776	44456	0.1 s	15.5 s

8 Previous Work on Product Configuration

In Sect. 6 we compared our approach to the CSP and DCSP formalisms. In this section we provide brief comparisons with several other approaches.

The generative CSP (GCSP) [7] approach introduces first-order constraints on activities of variables, on variable values and on resources. Constraints using arithmetic are also included. *Resources* are aggregate functions on intensionally defined sets of variables. They may restrict the set of variables active in a solution or generate new variables into a solution, thus providing a justification for the variables. In addition, a restricted form of DCSP activity constraints is used to provide justifications for activity of variables. **CRL** allows more expressive activity constraints than DCSP and a uniform representation of activity and other constraints. However, first-order rules, arithmetic and resource constraints are still missing from **CRL**.

Our approach fits broadly within the framework of constructive problem solving (CPS) [8]. In CPS the configurations are characterized as (possibly partial) Herbrand models of a theory in an appropriate logic language. The CPS approach does not require that elements in a configuration must have justifications but the need for a meta-level minimality criterion is mentioned.

Some implementations of configurators based on logic programming systems have been presented [15, 1]. In these approaches, similarly to our approach, a configuration domain oriented language is defined and the problem solving task is implemented on a variant of Prolog based on a mapping from the high-level language to Prolog. The languages are more complex and better suited for real modeling tasks. However, they are not provided a clear declarative semantics and the implementations use non-logical extensions of pure Prolog such as object-oriented Prolog and the cut. In contrast, we provide a simple declarative semantics and a sound and complete implementation for **CRL**.

9 Conclusions and Future Work

We have defined a rule-based language for representing typical forms of configuration knowledge, e.g., choices, dependencies between choices and incompatibilities. The language is provided with a declarative semantics based on a straightforward fix-point condition employing a simple transformation operator. The semantics induces formal definitions for the main concepts in product configuration, i.e., configuration models, requirements, configurations, valid condigurations and configurations that satisfy requirements. A novel feature of the semantics is that justifiability of a configuration (i.e., that each element in a configuration has a justication in terms of the configuration rules) is captured by Horn clause derivability but without resorting to a minimality condition on configurations. This approach has not been considered in previous work on product configuration. The semantics is closely related to well-known non-monotonic formalisms such as the stable model semantics [6] and the possible model semantics [14].

Avoiding minimality conditions in the semantics has a favorable effect on the complexity of the configuration tasks. The basic problems, i.e. validity of a configuration and whether a configuration satisfies a set of requirements, are polynomially decidable. This is important for practical configuration problems. It also implies that the other relevant decision problems are in **NP**.

We argue that the rule language is more expressive than constraints by showing that it cannot be modularly represented as CSP. The difficulty lies in capturing the justifications for a configuration using constraints. In addition, we show that the dynamic constraint satisfaction formalism can be embedded in our language but note that there is no obvious way of representing default negation and inclusive choices of **CRL** in that formalism.

There are indications that the proposed formal model provides a basis for solving practically relevant product configuration problems. An implementation of the rule language based on a translator to normal logic programs with the stable model semantics was tested on a small configuration problem. The results suggest that this approach is worth further research. Moreover, experiences in other domains show that efficient implementations of the stable model semantics are capable of handling tens of thousands of ground rules. Compiling a practically relevant configuration model from a high level representation into our language would seem to generate rule sets of approximately that size. Further research is needed to determine how our implementation scales for larger problems.

It may be possible to develop a more efficient algorithm that avoids the overhead incurred by the additional atoms and loss of information on the structure of the rules caused by the mapping to normal programs. Devising such an algorithm is an interesting subject of further work. A practically important task would be to identify additional syntactically restricted but still useful subsets of the language that would allow more efficient computation. Interactive product configuration where user makes hard decisions and computer only tractable ones may be the only feasible alternative for very large or complex problems. This type of configuration would be facilitated by devising polynomially computable approximations for valid configurations in **CRL**. Such approximations could also be used to prune the search space in an implemention of **CRL**.

It should be noted that the model does not adequately cover all the aspects of product configuration. Further work should include generalizing the rules to the first-order case, adding arithmetic operators to the language and defining constructs important for the domain such as optional choice directly in the language. These extensions are needed to conveniently represent resource constraints, attributes, structure and connections of components. Another important extension would be to define the notion of an optimal configuration (such as subset minimal, cardinality minimal or resource minimal configuration) and to analyze the complexity of optimality-related decision problems.

Acknowledgements. The work of the first author has been supported by the Helsinki Graduate School in Computer Science and Engineering (HeCSE) and the Technology Development Centre Finland and the work of the second author by the Academy of Finland through Project 43963. We thank Tommi Syrjänen for implementing the translation of **CRL** to normal logic programs.

References

1. T. Axling and S. Haridi. A tool for developing interactive configuration applications. *Journal of Logic Programming*, 19:658–679, 1994.
2. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*. Springer-Verlag, 1997.

3. J. Dix. Semantics of logic programs: Their intuitions and formal properties. In *Logic, Action and Information — Essays on Logic in Philosophy and Artificial Intelligence*, pages 241–327. DeGruyter, 1995.
4. 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.
5. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.
6. 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. The MIT Press, 1988.
7. A. Haselböck and M. Stumptner. An integrated approach for modelling complex configuration domains. In *Proceedings of the 13th International Conference on Expert Systems, AI, and Natural Language*, 1993.
8. R. Klein. A logic-based description of configuration: the constructive problem solving approach. In *Configuration—Papers from the 1996 AAAI Fall Symposium. Technical Report FS-96-03*, pages 111–118. AAAI Press, 1996.
9. X. Liu, C Ramakrishnan, and S. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19. Springer-Verlag, 1998.
10. J. McDermott. R1: a rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.
11. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 25–32. AAAI, MIT Press, 1990.
12. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303. The MIT Press, 1996.
13. I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429. Springer-Verlag, 1997.
14. C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13:145–172, 1994.
15. D. Searls and L. Norton. Logic-based configuration with a semantic network. *Journal of Logic Programming*, 8(1):53–73, 1990.
16. P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research report A47, Helsinki University of Technology, Helsinki, Finland, 1997. Available at http://saturn.hut.fi/pub/reports/A47.ps.gz.
17. T. Soininen and I. Niemelä. Formalizing configuration knowledge using rules with choices. Research report TKO-B142, Helsinki University of Technology, Helsinki, Finland, 1998. Presented at the Seventh International Workshop on Nonmonotonic Reasoning (NM'98), 1998.
18. J. Tiihonen, T. Soininen, T. Männistö, and R. Sulonen. State-of-the-practice in product configuration—a survey of 10 cases in the Finnish industry. In *Knowledge Intensive CAD*, volume 1, pages 95–114. Chapman & Hall, 1996.
19. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.