Representing Configuration Knowledge With Weight Constraint Rules*

Timo Soininen
TAI Research Centre
Helsinki Univ. of Technology
P.O.B. 9600, 02015 HUT
Finland
Timo.Soininen@hut.fi

Ilkka Niemelä
Lab. for Theoretical
Computer Science
Helsinki Univ. of Technology
P.O.B. 5400, 02015 HUT
Finland, Ilkka.Niemela@hut.fi

Juha Tiihonen
TAI Research Centre
Helsinki Univ. of Technology
P.O.B. 9600, 02015 HUT
Finland
Juha.Tiihonen@hut.fi

Reijo Sulonen
TAI Research Centre
Helsinki Univ. of Technology
P.O.B. 9600, 02015 HUT
Finland
Reijo.Sulonen@hut.fi

Abstract

The paper demonstrates that product configuration applications fit naturally the framework of answer set programming. It is shown that product configuration knowledge can be represented systematically and compactly using a logic program type rule language such that the answers of a configuration task, the configurations, correspond to the models of the rule representation. The paper presents such a rule-based formalization of a unified configuration ontology using a weight constraint rule language. The language extends normal logic programs with cardinality and weight constraints which leads to a compact and simple formalization. The complexity of the configuration task defined by the formalization is shown to be **NP**-complete.

Introduction

A *product configuration* task takes as input a configuration model consisting of a set of predefined component types and restrictions on combining the component types, as well as a set of customer requirements. The output is a *configuration*, a specification of a product individual that is correct with respect to the configuration model and satisfies the requirements. Knowledge-based systems for configuration tasks, *product configurators*, are an important application of AI techniques for companies selling products tailored to customer needs.

Product configuration fits nicely the framework of *answer set programming*. In a configuration task the configuration model and requirements can be seen as a theory and the answers to the task, the configurations, as models (or answer sets) satisfying the theory. The aim of this paper is to establish a systematic mapping from product configuration to answer set programming. We develop a general approach to translating configuration model knowledge and requirements into logic program type rules in such a way that the configurations correspond to the models of the rules.

A rule-based approach is employed because rules provide a natural way of representing configuration knowledge. As the semantical basis we use stable models (Gelfond & Lifschitz 1988) because they provide a groundedness property crucial for product configuration. The property states roughly that everything in a configuration must be grounded (or justified) by the configuration model. This is hard to capture in alternative approaches (see more details in the section on related work). However, we use an extension of normal logic programs (Niemelä, Simons, & Soininen 1999; Niemelä & Simons 2000). This language extends the propositional rule language of (Soininen & Niemelä 1999) to the first order case and adds constraints on sets of weighted literals to capture cardinality and resource constraints. This facilitates compact modeling based on a rich and practically relevant ontology. The language is implemented in the **Smodels** system and, hence, a prototype product configurator can be build on top of it with relatively little effort.

The rest of the paper is structured as follows. We first introduce briefly the extended rule language. Then, we show how a simplification of a generalized configuration ontology (Soininen *et al.* 1998) can be formalized using the rule language so that configurations correspond to stable models. Finally, we discuss the computational complexity of the configuration task defined in this way and finish by discussing related work.

Weight Constraint Rules

In this section we briefly introduce the weight constraint rule language used for formalizing configuration knowledge; for more information, see (Niemelä, Simons, & Soininen 1999; Simons 1999; Niemelä & Simons 2000). The language is equipped with *weight constraints* for representing weighted choices with lower and upper bounds and with *conditional literals* restricted by *domain predicates* to encode sets of atoms over which the choices are made.

First we consider ground rules where variables are not allowed. A *weight constraint rule*

$$C_0 \leftarrow C_1, \ldots, C_n \tag{1}$$

is built from *weight constraints* C_i of the form

$$l \leq \{a_1 = w_1, \ldots, a_n = w_n, \text{not } a_{n+1} = w_{n+1}, \ldots, \text{not } a_m = w_m\} \leq u \tag{2}$$

*This is a revised version of a paper presented at the ECAI00 Configuration Workshop, Aug 21–22 2000, Berlin, Germany.

where l and u are two numbers giving the lower and upper bound for the constraint, each a_i is a ground atomic formula and each w_i a number giving the weight for the corresponding literal (an atom or its negation). The semantics for such rules is given in terms of models that are sets of ground atoms. We say that a positive literal a is satisfied by a model S if $a \in S$ and a negative literal not a if $a \notin S$. A weight constraint is satisfied by a model S if the sum of weights of the literals satisfied by S is between the bounds l and u. For example,

$$30 \leq \{\ a = 10, b = 20, \text{not } c = 30, \text{not } d = 35\ \} \leq 40$$

is satisfied by a model $\{a, b, c, d\}$ for which the sum of the weights is $10 + 20 = 30$ but not by a model $\{a, b, c\}$ for which the sum is 65. A missing lower bound is taken to be $-\infty$ and a missing upper bound ∞. Note that "$<$" could also be similarly used in a constraint.

We use shorthands for some special cases of weight constraints. A *cardinality constraint* where all weights are 1 is written as

$$l\ \{a_1, \ldots, a_n, \text{not } a_{n+1}, \ldots, \text{not } a_m\}\ u \tag{3}$$

and a cardinality constraint $1\{p\}$ simply as a literal p. We call rules where all constraints C_i are literals *normal rules*. We can also write mixed rules such as

$$0\ \{a, b\}\ 1 \leftarrow c\ . \tag{4}$$

The semantics of a set of rules is captured by *stable models* fulfilling two properties: (i) a stable model *satisfies* the rules and (ii) is *justified* by the rules. A rule of form (1) is satisfied by a model S iff S satisfies the *head* C_0 whenever it satisfies each of the *body* constraints $C_1, \ldots, C_n$. A rule without a head is satisfied if at least one of the body constraints is not. We explain the basic idea of justifiability through an example. Consider rule (4) satisfied, e.g., by a model $\{a\}$. This model is not justified by the rule since the body of (4) enabling the choice of a, is not justified. Indeed, for (4) the only stable model is the empty set. Suppose we add a fact $c \leftarrow$. Then each stable model of the two rules contains c. The rules have three stable models $\{c\}$, $\{c, a\}$, and $\{c, b\}$ as the body of (4) is justified by the new fact.

For compact and structured representation and ease of maintenance, it is very useful to provide rules with variables. The semantics for rules with variables (Niemelä, Simons, & Soininen 1999) is obtained by considering the ground instantiation of the rules with respect to their Herbrand universe. However, in this general case the rule language is highly undecidable. A decidable subclass is obtained by considering the function-free and *domain-restricted* case (Niemelä, Simons, & Soininen 1999). Here each variable has a domain predicate in the body of the rule providing the domain over which the variable ranges. Domain predicates can be defined using non-recursive normal rules starting from basic ground facts. The rules defining domain predicates have a unique stable model. By the domain-restrictedness of the rules, this model provides the domains of the variables in all the rules.

In order to compactly represent sets of literals in constraints, *conditional literals*, i.e., expressions such as $part(X, D) : ide(D)$, can be used in place of literals where the conditional part $ide(D)$ is a domain predicate (or conjunction of them). When using conditional literals we need to distinguish between *global* variables (quantifying universally over the whole rule) and *local* ones (with a scope of a single conditional literal). The distinction is made implicitly by using the following convention: a variable is local to a conditional literal if it appears only in this literal in the rule and all other variables are global in the rule. For example, in the rule

$$1\ \{part(X, D) : ide(D)\}\ 2 \leftarrow pc(X) \tag{5}$$

the variable D is local to the conditional literal in the head but X is a global variable. A rule with conditional literals is understood as a shorthand for a set of ground rules resulting from a two step process. First all global variables are replaced by ground terms satisfying the domain predicates in every possible way giving a set of rules where variables appear only as local ones in conditional literals. Each such literal corresponds to a sequence of ground instances of the main predicate such that the domain predicate in the conditional part holds. For example, consider rule (5). By domain-restrictedness, predicates $ide(D)$ and $pc(X)$ are domain ones and defined by non-recursive rules. Now (5) corresponds to a set of ground rules

$$1\ \{part(pc_i, d_1), \ldots, part(pc_i, d_m)\}\ 2 \leftarrow pc(pc_i) \tag{6}$$

where X is replaced by a constant pc_i for which $pc(pc_i)$ holds and where the resulting conditional literal $part(pc_i, D) : ide(D)$ corresponds to a sequence of ground facts $part(pc_i, d_j)$ for which $ide(d_j)$ holds.

An implementation of the weight constraint rule language called **Smodels** is publicly available at `http://www.tcs.hut.fi/Software/smodels/`. The current implementation supports only integer weights in order to avoid complications due to finite precision of standard implementations of real number arithmetic.

Configuration Knowledge

In this section we show how to represent configuration knowledge using the rule language. We distinguish between the rules giving *ontological definitions* and the rules representing a configuration model. The former are not changed when defining a configuration model of a product and are enclosed in a box in the following. The latter appear only in the examples and are product specific. We further make the convention that the domain predicates are typeset normally whereas other predicates are typeset in ***boldface***.

The representation is based on a simplified version of a general configuration ontology (Soininen *et al.* 1998). In the ontology, there are three main categories of knowledge: *configuration model knowledge*, *requirements knowledge* and *configuration solution knowledge*. Configuration model knowledge specifies the entities that can appear in a configuration and the rules on how the entities can be combined. More specifically, a configuration model represented according to the simplified ontology consists of (i) a set of component types, (ii) a set of resource types, (iii) a set of port types, (iv) a taxonomy or class hierarchy of types, (v) compositional structure of component types, (vi) resource production and use by component types, (vii) connections between ports of components types, and (vii) a set of constraints.

In our approach, a configuration model is represented as a set of rules. Configuration solution knowledge specifies a configuration, defined as a stable model of the set of rules in the configuration model. These definitions correspond to the usual answer set programming paradigm. However, for configuration tasks the requirements knowledge, represented as another set of rules, has a different status from that of the configuration model. The requirements must also be satisfied by a configuration but cannot justify any elements in it.

Types, Individuals and Taxonomy

Most approaches to configuration distinguish between *types* and *individuals*, often called classes and instances. Types in a configuration model define intensionally the properties, such as parts, of their individuals that can appear in a configuration. In the simplified ontology, a configuration model includes the following disjoint sets of types: *component types*, *resource types*, and *port types*. They are organized in a *taxonomy* or *class hierarchy* where a subtype *inherits* the properties of its supertypes in the usual manner. For simplicity we require that each individual in a configuration is of a concrete, i.e. leaf, type.

Individuals of concrete component types are naturally represented as object constants with unique names. This allows several individuals of the same type in a configuration. Types are represented by unary domain predicates ranging over their individuals. Since a resource of a given type need not be distinguished as an individual, there is exactly one individual of each concrete resource type. The individuals c_i in a configuration are represented by the predicate $\boldsymbol{in}(c_i)$.

The type predicates are used as conditional parts of literals to restrict the applicability of the rules to individuals of the type only. This facilitates defining properties of individuals (see below). The taxonomy is represented using rules stating that individuals of a type are also individuals of its supertypes, which enables monotonic inheritance of the property definitions.

Example 1. A computer is represented by component type pc (PC), which is a subtype of cmp, the generic component type. For each individual pc_i of component type pc in a configuration it holds that $pc(pc_i)$. Component type ide (IDE device) is also a subtype of cmp. A type representing IDE hard disks, hd, is a subtype of ide. Actual hard disks are represented as subtypes of hd, namely hd_a and hd_b. IDE CD ROM devices cd_a and cd_b are subtypes of type cd, which is a subtype of ide. Software packages are represented by type sw. Software package types sw_a and sw_b are subtypes of sw. Resource and port types are introduced in the following sections. The following rules define the component types and their taxonomy:

$$
\begin{array}{lll}
cmp(C) \leftarrow pc(C) & ide(C) \leftarrow hd(C) & hd(C) \leftarrow hd_a(C) \\
cmp(C) \leftarrow ide(C) & ide(C) \leftarrow cd(C) & hd(C) \leftarrow hd_b(C) \\
cmp(C) \leftarrow sw(C) & sw(C) \leftarrow sw_a(C) & cd(C) \leftarrow cd_a(C) \\
 & sw(C) \leftarrow sw_b(C) & cd(C) \leftarrow cd_b(C)
\end{array}
$$

Compositional Structure

The decomposition of a product to its parts, referred to as *compositional structure*, is an important part of configuration knowledge. A component type defines its direct parts through a set of *part definitions*. A part definition specifies a *part name*, a *set of possible part types* and a *cardinality*. The part name identifies the role of a component individual as a part of another. The possible part types indicate the component types whose component individuals are allowed in the part role. The cardinality, an integer range, defines how many component individuals must occur as parts in the part role.

For simplicity, we assume that there is exactly one independent component type, referred to as *root type* of the compositional structure formed by the part definitions. An individual of this type is the root of the product structure. In a configuration there is exactly one individual of the root type of the compositional structure. Component types are also for simplicity assumed to be exclusive, i.e. a component individual is part of at most one component individual. Further, a component individual must not be directly or transitively a part of itself. These restrictions are placed to prevent counterintuitive structures of physical products. In effect the compositional structure of a configuration forms a tree of component individuals, and each component individual in a configuration is in the tree.

The fact that a component individual c_1 has as a part another component individual c_2 with a part name pn is represented by the predicate $\boldsymbol{pa}(c_1, c_2, pn)$. A part name is thus represented as an object constant and the set of part names pn_i in a configuration model are captured using the domain predicate $pan(pn_i)$.

A part definition is represented as a rule that employs a cardinality constraint in the head. The individuals c_j of possible part types in a part definition with part name pn of a component type whose individuals are c_i are represented using the domain predicate $ppa(c_i, c_j, pn)$. It is defined as the union of the individuals of the possible component types.

Example 2. The root component type pc has as its parts 1 to 2 mass-storage units (with part name ms) of type ide, and 0 to 10 optional software packages (with part name swp) of type sw. Note that using an abstract (non-leaf) type, such as ide, in a part definition effectively enables a choice from its concrete subtypes. The fact that pc is the root type of the compositional structure and the part names and possible part types of PC are represented as follows:

$$\begin{array}{ll} root(C) \leftarrow pc(C) & \\ pan(ms) \leftarrow & ppa(C_1, C_2, ms) \leftarrow pc(C_1), ide(C_2) \\ pan(swp) \leftarrow & ppa(C_1, C_2, swp) \leftarrow pc(C_1), sw(C_2) \end{array}$$

The part definitions for the mass storage and software package roles are represented as follows:

$$\begin{array}{r} 1\ \{\boldsymbol{pa}(C_1, C_2, ms) : ppa(C_1, C_2, ms)\}\ 2 \leftarrow \boldsymbol{in}(C_1), \\ pc(C_1) \\ 0\ \{\boldsymbol{pa}(C_1, C_2, swp) : ppa(C_1, C_2, swp)\}\ 10 \leftarrow \boldsymbol{in}(C_1), \\ pc(C_1) \end{array}$$

The ontological definitions that exactly one individual of the root type of the compositional structure is in a configuration, and that other component individuals are in a configuration if they are parts of something are given as follows:

$$\boxed{\begin{array}{rcl} 1\ \{\boldsymbol{in}(C) : root(C)\}\ 1 & \leftarrow & \\ \boldsymbol{in}(C_2) & \leftarrow & \boldsymbol{pa}(C_1, C_2, N), cmp(C_1), \\ & & cmp(C_2), pan(N) \end{array}}$$

The exclusivity of component individuals is captured by the following ontological definition that a component individual can not be a part of more than one component individual:

$$\boxed{\leftarrow cmp(C_2), 2\ \{\boldsymbol{pa}(C_1, C_2, N) : cmp(C_1) : pan(N)\}}$$

As the structure must be a tree, the following ontological definitions of the transitive closure of *ppa* and its asymmetricity are needed:

$$\boxed{\begin{array}{l} \boldsymbol{ppa}_t(C_1, C_2) \leftarrow ppa(C_1, C_2, N) \\ \boldsymbol{ppa}_t(C_1, C_3) \leftarrow ppa(C_1, C_2, N), cmp(C_3), \\ \qquad\qquad\qquad \boldsymbol{ppa}_t(C_2, C_3) \\ \qquad\qquad\quad \leftarrow \boldsymbol{ppa}_t(C, C), cmp(C) \end{array}}$$

Resources

The resource concept is useful in configuration for modeling the production and use of some more or less abstract entity, such as power or disk space. Some component individuals produce resources and other component individuals use them. The amount produced must be greater than or equal to the amount used.

A component type specifies the resource types and amounts its individuals produce and use by *production definitions* and *use definitions*. Each production or use definition specifies a *resource type* and a *magnitude*. The magnitude specifies how much of the resource type component individuals produce or use.

A resource type is represented as a domain predicate. Only one resource individual with the same name as the type is needed, since a resource is not a countable entity. A production and a use definition of a component type is represented using the domain predicate $prd(c, r, x)$ where c is a component individual of the producing or using component type, r the individual of the produced or used resource type and x the magnitude. Use is represented as negative magnitude.

Example 3. Disk space is used by the software packages and produced by hard disks. Disk space is represented by resource type *ds*. Each subtype of type *sw* uses a fixed amount of disk space, represented by their use definitions: sw_a uses 400MB and sw_b 600 MB. Hard disks of type hd_a produce 700MB and of type hd_b 1500MB of *ds*. The following rules represent the *ds* resource type and the production and use definition of component types:

$$\begin{array}{rr} res(R) \leftarrow ds(R) & prd(C, ds, -400) \leftarrow sw_a(C) \\ ds(ds) \leftarrow & prd(C, ds, -600) \leftarrow sw_b(C) \\ prd(C, ds, 700) \leftarrow hd_a(C) & prd(C, ds, 1500) \leftarrow hd_b(C) \end{array}$$

The production and use of a resource type by the component individuals is represented as weights of the predicate $\boldsymbol{in}()$. The ontological definition that the resource use must be satisfied by the production is expressed with a weight constraint rule stating that the sum of the produced and used amounts must be greater than or equal to zero:

$$\boxed{\leftarrow res(R), \{\boldsymbol{in}(C) : prd(C, R, M) = M\} < 0}$$

Ports and Connections

In addition to hierarchical decomposition, it is often necessary to model connections or compatibility between component individuals. A *port type* is a definition of a connection interface. A *port individual* represents a "place" in a component individual where at most one other port individual can be connected. A port type has a *compatibility definition* that defines a set of port types whose port individuals can be connected to port individuals of the port type.

A component type specifies its connection possibilities by *port definitions*. A port definition specifies a *port name*, a port type and *connection constraints*. Port individuals of the same component individual cannot be connected to each other. For simplicity, we consider only a limited connection constraint specifying whether a connection to a port individual is obligatory or optional. Effectively an obligatory connection sets a requirement for the existence of and connection with a port of a compatible component individual.

Port types are represented as domain predicates and port individuals as uniquely named object constants. Compatibility of port types is represented as the domain predicate $cmb(p_1, p_2)$, where p_1 and p_2 are port individuals of compatible port types and a rule that any two compatible port individuals can be connected. A connection between port individuals p_1 and p_2 are represented as the symmetric, irreflexive predicate $\boldsymbol{cn}(p_1, p_2)$. A port individual is connected to at most one other port individual. The following rules represent these ontological definitions:

$$\boxed{\begin{array}{r} 0\ \{\boldsymbol{cn}(P_1, P_2)\}\ 1 \leftarrow \boldsymbol{in}(P_1), \boldsymbol{in}(P_2), cmb(P_1, P_2) \\ \boldsymbol{cn}(P_2, P_1) \leftarrow \boldsymbol{cn}(P_1, P_2), prt(P_1), prt(P_2) \\ \leftarrow prt(P_1), 2\ \{\boldsymbol{cn}(P_1, P_2) : prt(P_2)\} \\ \leftarrow prt(P_1), \boldsymbol{cn}(P_1, P_1) \end{array}}$$

Example 4. The configuration model includes port types ide_c and ide_d that are subtypes of the general port type prt. These types represent the computer and peripheral device sides of an IDE connection. The compatibility definition of ide_c states that it is compatible with ide_d. Correspondingly ide_d states compatibility with ide_c. The following rules represent the port types and compatibility definitions:

$$\begin{array}{ll} prt(P) \leftarrow ide_c(P) & cmb(P_1,P_2) \leftarrow ide_c(P_1), ide_d(P_2) \\ prt(P) \leftarrow ide_d(P) & cmb(P_1,P_2) \leftarrow ide_d(P_1), ide_c(P_2) \end{array}$$

A port definition of a component type is represented as a rule very similar to a part definition, but with the predicate $\boldsymbol{po}(c_1, p_1, pn)$ signifying that a component individual c_1 has a port individual p_1 with the port name pn. The $pon(pn_i)$ domain predicate captures the set of port names pn_i.

Example 5. Component type pc has two ports with names ide_1 and ide_2 of type ide_c for connecting IDE devices. Component type ide has one port of type ide_d, called ide_p, for connecting the device to a computer. The ide_p port has a connection constraint that connection to that port is obligatory. In rule form:

$$pon(ide_1) \leftarrow \quad pon(ide_2) \leftarrow \quad pon(ide_p) \leftarrow$$

$$\begin{array}{l} 1\,\{\boldsymbol{po}(C,P,ide_1) : ide_c(P)\}\,1 \leftarrow \boldsymbol{in}(C), pc(C) \\ 1\,\{\boldsymbol{po}(C,P,ide_2) : ide_c(P)\}\,1 \leftarrow \boldsymbol{in}(C), pc(C) \\ 1\,\{\boldsymbol{po}(C,P,ide_p) : ide_d(P)\}\,1 \leftarrow \boldsymbol{in}(C), ide(C) \end{array}$$

$$\begin{array}{l} \leftarrow ide(C), ide_d(P_1), \boldsymbol{po}(C,P_1,ide_p), \\ \quad \{\boldsymbol{cn}(P_1,P_2) : prt(P_2)\}\,0 \end{array}$$

The ontological definitions that a port individual is in a configuration if some component individual has it and that port individuals of one component individual cannot be connected are also needed:

$$\boxed{\begin{array}{l} \boldsymbol{in}(P) \leftarrow cmp(C), pon(N), prt(P), \boldsymbol{po}(C,P,N) \\ \quad\quad \leftarrow cmp(C), pon(N_1), prt(P_1), \boldsymbol{po}(C,P_1,N_1), \\ \quad\quad\quad pon(N_2), prt(P_2), \boldsymbol{po}(C,P_2,N_2), \boldsymbol{cn}(P_1,P_2) \end{array}}$$

Constraints

All approaches to configuration have some kinds of *constraints* as a mechanism for defining the conditions that a correct configuration must satisfy. Rules without heads can be used in our approach to represent typical forms of constraints.

Example 6. A PC configuration model could require that a hard disk of type hd must be part of a PC:

$$\leftarrow pc(C_1), \{\boldsymbol{pa}(C_1,C_2,N) : hd(C_2) : pan(N)\}\,0$$

Computational Complexity

For the complexity analysis of configuration tasks we make the following assumptions. A configuration model $\mathcal{CM}$ represented according to the ontology is translated to a set of rules CM as above including the rules for ontological definitions. Then, a set of ground facts S is added, providing the individuals that can be in a configuration. Set S is constructed out of the domain predicates representing the concrete types in CM and unique object constants for the individuals of concrete types. The set of rules $CM \cup S$ is all that is needed for representing the product, and subsequently configuring it. The requirements are represented using another set of rules R. The set S can be thought of as a storage of individuals from which a configuration is to be constructed. It thus induces an upper bound on the size of a configuration. This is important since if such a bound cannot be given, the configurations could in principle be arbitrarily large, even infinite.

We further make the assumption that the number of variables in the rule representation of each constraint in $\mathcal{CM}$ and each rule in R is bounded by some constant. This is based on the observation that even checking whether a constraint rule or requirement rule of arbitrary length is satisfied by a configuration is computationally very hard. This would be contrary to the intuition that checking whether a constraint or requirement is in effect in a configuration should be easy. Since the ontological definitions in CM have a limited number of variables, this assumption implies that there is a bound c on the number of variables in any rule of CM and R.

The above assumptions lead to the following definition of the decision version of the configuration task:

Definition 7. CONFIGURATION TASK(D): Given a configuration model $\mathcal{CM}$ translated to a set of rules CM, a set of ground facts S, and a set of rules R, where the number of variables in any rule of CM and R is bounded by a constant c, is there a configuration C (a stable model of $CM \cup S$) such that C satisfies R?

Theorem 8. *(Soininen* et al. *2000) CONFIGURATION TASK(D) is* ***NP****-complete in the size of* $CM \cup S \cup R$.

Discussion and Related Work

There are several approaches that define a configuration oriented language, a mapping to an underlying language, and implement the configuration task using an implementation of the underlying language. The main distinction and advantage provided by the answer set programming approach taken in this work is the groundedness property of solutions.

The groundedness property of the semantics of weight constraint rules extends the similar property of stable model semantics (Gelfond & Lifschitz 1988). When applied to the configuration domain it captures formally the idea that a correct configuration must not contain anything that is not justified by the configuration model. There is no need to add completion or frame axioms to accomplish this. Capturing groundedness in the semantics thus allows a more compact and modular representation of configuration knowledge. This also implies that if a part of a configuration model or the formalization of the conceptualization is changed, only those rules that represent the changed part need to be modified to capture the new configuration model or formalization.

This is in contrast to approaches based on a monotonic formalism such as CSP or classical logic. For example, in (Friedrich & Stumptner 1999) a form of classical predicate logic is used. The mapping of configuration knowledge to predicate logic is not modular, in the sense that the types, property definitions and constraints are not formalized independently. Several completion axioms are required to ensure that a configuration is complete and does not contain extraneous things. In fact, it can be shown that even a simple special case of propositional weight constraint rules cannot be represented modularly by CSP or propositional logic (Soininen & Niemelä 1999).

A further aspect of groundedness is that it can be used to clearly distinguish between the roles of the configuration model and requirements. Without the groundedness property, one must resort to meta-level conditions. For example, elements other than in a configuration model must be excluded from the configuration even if the requirements state their existence (Friedrich & Stumptner 1999). By separating the notions of a configuration satisfying a set of rules and being grounded by the set of rules, one obtains the following clear distinction: a suitable configuration must be correct, i.e. satisfy and be grounded by a configuration model, but only needs to satisfy the requirements. Thus, requirements cannot add unwanted elements into a configuration.

The groundedness of configurations has been identified as important in, e.g., the research on dynamic constraint satisfaction problems (DCSP) (Mittal & Falkenhainer 1990) and the rule based approaches to configuration in (Soininen & Niemelä 1999; Syrjänen 2000). In (Soininen, Gelle, & Niemelä 1999), DCSP is given a new semantics using similar ideas as for weight constraint rules and extended with a simple form of cardinality constraints. It is also shown that DCSP can be translated in a straightforward manner to weight constraint rules and thus implemented using **Smodels**. In (Soininen & Niemelä 1999; Syrjänen 2000) two rule languages are defined and used to capture configuration knowledge based on simpler ontologies consisting of components, choices and dependencies. These languages turn out to be special cases of weight constraint rules and directly implementable on top of **Smodels** (Soininen 2000; Syrjänen 2000).

DCSP does not allow straightforward formalization of the configuration knowledge addressed in this paper as it lacks constructs for directly representing objects and domain and other predicates and for defining new predicates on the basis of existing ones. The same difficulty holds also for the propositional rule language in (Soininen & Niemelä 1999). The rule language in (Syrjänen 2000) does support predicates and their definitions. However, neither it or the language in (Soininen & Niemelä 1999) supports weight and cardinality constraints needed for compact configuration knowledge representation.

When compared to the previous rule based approaches, we handle in this paper a richer ontology covering the practically important aspects of structure, connections and resource interactions. However, some central issues such as attribute modeling, constructs for supporting optimization, and arithmetic expressions remain as further extensions. It should be noted that attribute definitions with relatively small integer or enumerated domains and arithmetic on integers can be supported by **Smodels** through weight constraint rules and built-in predicates (Niemelä, Simons, & Soininen 1999). In addition, the weight constraints provide a way of expressing decision versions of optimization tasks with a given bound. This mechanism can be extended to handle optimization criteria.

Several other logic programming languages with nonmonotonic semantics could conceivably be used for configuration knowledge representation. However, most declarative semantics and forms of logic programs are rather inadequate for capturing many important aspects of configuration knowledge. They provide limited means only, e.g. disjunctions, for expressing choices among alternatives. Little support is provided for expressing cardinality or weight constraints on choices. For most logic program semantics with disjunctive choices, the models of programs are required to be minimal instead of grounded in the sense of weight constraint rules. In our view subset minimality is in fact a specific optimality criterion which is not very relevant for configuration applications where measuring, e.g., cost and resource consumption is more important. Such optimality criteria can be added on top of the semantics of weight constraint rules whose role is to provide the correct solutions.

However, there are semantics allowing non-minimal models and, in fact, if we consider the subclass of weight constraint rules with a simpler form of choices in the heads of rules and default negation (Soininen & Niemelä 1999), the semantics coincides with the possible models for disjunctive programs (Sakama & Inoue 1994). This class of programs does not, however, include cardinality or weight constraints. Other logic program approaches have also associated numbers with the atoms in rules, or with the rules. However, the numbers are usually interpreted as priorities, preferences, costs, probabilities or certainty factors of rules (see e.g. (Brewka & Eiter 1998; Buccafurri, Leone, & Rullo 1997) and the references there). The NP-SPEC (Cadoli *et al.* 1999) approach is also somewhat similar in aims. It is not based on a generalization of the stable model semantics, however, but on circumscription, and it does not treat choices, weight constraints and rules uniformly.

There are also some approaches to configuration based on logic programming systems, e.g. (Axling & Haridi 1994). In these approaches, the configuration task is implemented on a variant of Prolog based on a mapping from a high-level language to Prolog. However, the languages 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. Several authors have also proposed using constraint logic programs (CLP) to represent configuration knowledge and implement the configuration task (e.g. (Sharma & Colomb 1998)). However, the CLP languages in these approaches are not based on a semantics with a groundedness property and do not include default negation or choices through disjunction or cardinality constraints.

Conclusions and Future Work

We have presented an approach to formally representing product configuration knowledge and implementing the product configuration task on the basis of the answer set programming paradigm. We defined a unified and detailed ontology for representing configuration knowledge on different aspects of products. The ontology and configuration models are formalized by defining a mapping to a new type of logic programs, weight constraint rules, designed for representing configuration knowledge uniformly and in a straightforward manner. The language allows a compact and simple formalization. The complexity of the configuration task defined by the formalization is shown to be **NP**-complete.

However, the language does not allow real number arithmetic. Extending the language and its implementation with this and optimization related constructs and formalizing a more extensive configuration ontology are important subjects of further work. In addition, the computational complexity of different possible conceptualizations should be further analyzed, and the implementation performance should be tested.

Acknowledgements

Helsinki Graduate School in Computer Science and Engineering has funded the work of the first author, Technology Development Centre Finland the work of the first and third authors, and Academy of Finland (Project 43963) the work of the second author.

References

Axling, T., and Haridi, S. 1994. A tool for developing interactive configuration applications. *J. of Logic Programming* 19:658–679.

Brewka, G., and Eiter, T. 1998. Preferred answer sets for extended logic programs. In *Principles of Knowledge Representation and Reasoning Proceedings of the Sixth International Conference*, 86–97.

Buccafurri, F.; Leone, N.; and Rullo, P. 1997. Strong and weak constraints in disjunctive datalog. In *Proc. of the 4th Int. Conference on Logic Programming and Non-Monotonic Reasoning*, 2–17.

Cadoli, M.; Palopoli, L.; Schaerf, A.; and Vasile, D. 1999. NP-SPEC: An executable specification language for solving all problems in NP. In *Practical Aspects of Declarative Languages, LNCS 1551*, 16–30.

Friedrich, G., and Stumptner, M. 1999. Consistency-based configuration. In *Configuration. AAAI Technical Report WS-99-05*, 35–40.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the 5th International Conference on Logic Programming*, 1070–1080.

Mittal, S., and Falkenhainer, B. 1990. Dynamic constraint satisfaction problems. In *Proc. of the 8th Nat. Conf. on AI (AAAI90)*, 25–32.

Niemelä, I., and Simons, P. 2000. Extending the Smodels system with cardinality and weight constraints. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Kluwer Academic Publishers.

Niemelä, I.; Simons, P.; and Soininen, T. 1999. Stable model semantics of weight constraint rules. In *Proc. of the Fifth Intern. Conf. on Logic Programming and Nonmonotonic Reasoning*, 317–331.

Sakama, C., and Inoue, K. 1994. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning* 13:145–172.

Sharma, N., and Colomb, R. 1998. Mechanizing shared configuration and diagnosis theories through constraint logic programming. *Journal of Logic Programming* 37:255–283.

Simons, P. 1999. Extending the stable model semantics with more expressive rules. In *Proc. of the Fifth Intern. Conf. on Logic Programming and Nonmonotonic Reasoning*, 305–316.

Soininen, T., and Niemelä, I. 1999. Developing a declarative rule language for applications in product configuration. In *Proc. of the First Int. Workshop on Practical Aspects of Declarative Languages*, 305–319.

Soininen, T.; Tiihonen, J.; Männistö, T.; and Sulonen, R. 1998. Towards a general ontology of configuration. *AI EDAM* 12:357–372.

Soininen, T.; Niemelä, I.; Tiihonen, J.; and Sulonen, R. 2000. Unified configuration knowledge representation using weight constraint rules. Research report TKO-B149, Helsinki University of Technology, Laboratory of Information Processing Science.

Soininen, T.; Gelle, E.; and Niemelä, I. 1999. A fixpoint definition of dynamic constraint satisfaction. In *Proc. of the 5th International Conf. on Principles and Practice of Constraint Programming*, 419–433.

Soininen, T. 2000. *An approach to knowledge representation and reasoning for product configuration tasks*. Ph.D. Dissertation, Helsinki University of Technology, Finland.

Syrjänen, T. 2000. Including diagnostic information in configuration models. In *Proc. of the First International Conference on Computational Logic, Automated Deduction: Putting Theory into Practice*, 837–851.