Unified Configuration Knowledge Representation Using Weight Constraint Rules

Timo Soininen[1] and **Ilkka Niemelä**[2] and **Juha Tiihonen**[1] and **Reijo Sulonen**[1]

Abstract. In this paper we present an approach to formally defining different conceptualizations of configuration knowledge, modeling configuration knowledge, and implementing the configuration task on the basis of the formalization. The approach is based on a weight constraint rule language and its efficient implementation designed for representing different aspects of configuration knowledge uniformly and compactly. The use of the language to represent configuration knowledge is demonstrated through formalizing a unified configuration ontology. The language allows a compact and simple formalization. The complexity of the configuration task defined by the formalization is shown to be **NP**-complete.

1 Introduction

Product configuration can be roughly defined as the task of producing a specification of a product individual, a *configuration*, from a set of predefined component types while taking into account a set of restrictions on combining the component types. Knowledge based systems for configuration tasks, *product configurators*, have recently become an important application of artificial intelligence techniques for companies selling products tailored to customer needs.

Several approaches to configuration are based on relatively well-understood general formalisms, such as constraint satisfaction, its extensions, and variants of description logics. Other approaches have defined specific configuration domain oriented conceptual foundations. These include the three main conceptualizations of configuration knowledge as resource balancing, product structure and connections within a product [11]. Despite the research, there are few formal models of configuration aiming to unify the different formal and conceptual approaches. Such models are needed to facilitate rigorous analysis and comparison of the different approaches.

In this paper we present an approach that facilitates formal representation of configuration knowledge based on different and unified conceptualizations. In contrast to many previous approaches that have developed special purpose algorithms for each conceptualization, our aim is to provide for fast prototyping of conceptualizations and configuration tasks based on them. This is accomplished by capturing the configuration knowledge using a neutral representation language called weight constraint rules and implementing the configuration task using a general implementation of such rules. Therefore, when changing the configuration model or its underlying conceptual foundation, only the representation of the knowledge is changed. There is no need to design a new algorithm for the configuration task, as the general implementation of the rule language can still be used.

We first briefly introduce the logic program like rule language that has been specifically designed to allow representing knowledge on different aspects of configuration in a uniform and simple manner. At the same time, the computational efficiency of the configuration task has also been taken into account. The language has a declarative semantics providing a formal definition for the product configuration task. The semantics captures the property that a configuration can only contain elements that are *justified* by the configuration model. This property has been identified as important in e.g. the research on dynamic constraint satisfaction problems (DCSP) [5]. The justification property also allows a more compact and modular representation of knowledge than e.g. classical logic or constraint satisfaction problems [10, 9]. For example, a subset of the rule language can represent e.g. CSP and DCSP in a simple, compact manner [10]. It also extends these approaches with *cardinality* and *weight constraints*.

After introducing the rule language we demonstrate its applicability to configuration knowledge representation. This is done by showing how a configuration model represented using a simplification of a generalized configuration ontology [11] covering product structures, resource balancing, connections and constraints is mapped to a set of rules. The rules are not used by product modelers but as an intermediate language that a higher level configuration modeling language is translated to. Due to the built-in justification property of the language, the mapping is modular and relatively simple. Extending the representation to cover a more complex conceptualization is straightforward, although we discuss in brief some challenges to this. The relevant decision problem for the configuration task is defined and shown to be **NP**-complete on the basis of the mapping to rules. Finally, we discuss our approach in relation to previous similar work.

2 Weight Constraint Rules

In this section we briefly introduce the weight constraint rule language used for formalizing configuration knowledge (for more information on the language see [7, 8]). The language extends logic programs by offering support for expressing choices as well as cardinality and resource constraints that are often useful in configuration:

Example 1. A (partial) configuration model of a simplified PC could consist of the following knowledge: There is a known set of different types of IDE hard disks, IDE CD ROMs, scanners and SCSI cards that can be parts of a PC. The different IDE hard disks and IDE CD ROMs are IDE devices. A PC must have from one to two IDE devices, of which at least one must be an IDE hard disk. In addition, a desktop publishing PC must have a scanner, which can be a flatbed or a slide scanner. As the slide scanner is a SCSI device, including it in

[1] Helsinki University of Technology, TAI Research Center and Lab. of Information Processing Science, P.O.Box 9555, FIN-02015 HUT, Finland, email: {Timo.Soininen,Juha.Tiihonen,Reijo.Sulonen}@hut.fi

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

a configuration requires a SCSI card. There are two different types of SCSI cards to select from. Furthermore, each component has an associated price. The customer might require that the total price, summed over the components in a configuration, must be less than $900.

The rule language aims to capture the type of configuration knowledge in the example in a compact form. To do this, it is equipped with *conditional literals* restricted by *domain predicates* for representing the different types of components and other objects and the knowledge on them, such as the IDE devices above. *Cardinality constraints* are introduced for representing choices with lower and upper bounds, such as for the IDE devices in the above example. Resource constraints on a configuration exemplified by the prices of the components above are captured in the generalized language using *weight constraints*, which give the language its name.

First we consider ground rules, i.e., rules where variables quantifying over a rule are not allowed. Then we introduce rules with variables. A *weight constraint rule*

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

is built from *weight constraints* C_i of form

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

where L and U are two numbers giving the lower and upper bound for the constraint, each a_i is an atomic formula and each w_i a number giving the weight for the corresponding literal (an atom or its negation). Such a rule is intuitively read as follows: if the *body*, i.e. each of C_1 to C_n, holds then the *head* C_0 must also hold. 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 $\text{not } a$ is satisfied 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. Note that "<" could also be similarly used in a constraint. E.g.,

$$30 \leq \{ \ part(pc_1, d_1) = 10, part(pc_1, d_2) = 20, \\ part(pc_1, d_3) = 30, part(pc_1, d_4) = 30 \ \} \leq 40$$

is satisfied by a model $\{part(pc_1, d_1), part(pc_1, d_2)\}$ for which the sum of the weights is $10 + 20 = 30$ but not by a model $\{part(pc_1, d_3), part(pc_1, d_4)\}$ for which the sum is 60.

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 $1\{l\}1$ with one literal and 1 as both bounds simply as a literal l. We call rules where all constraints C_i are literals *normal rules*. We can also write mixed rules such as

$$1 \ \{part(pc_1, sc_1), part(pc_1, sc_2)\} \ 1 \leftarrow part(pc_1, sscan) \tag{4}$$

The semantics of a set of rules is captured by a subclass of models called *stable models*. They fulfill two important properties: (i) a stable model *satisfies* the rules and (ii) is *justified* by the rules. A rule r of form (1) is satisfied by a model S iff S satisfies C_0 at least whenever it satisfies each of $C_1, \ldots, C_n$. A rule without a head is satisfied if at least one of the body constraints is not. The technical details of the justifiability property can be found in [7], but we explain the basic intuition through an example. Consider rule (4). It is satisfied, e.g., by the model $\{part(pc_1, sc_1)\}$. However, this model is not justified by the rule since there is no reason to include the head of the rule in the model, since the body of the rule, $part(pc_1, sscan)$, is not justified. Indeed, for the rule (4) the only stable model is the empty set which also satisfies the rule. Suppose we add two rules

$$1 \ \{part(pc_1, fscan), part(pc_1, sscan)\} \ 2 \quad \leftarrow \quad dtp(pc_1) \tag{5}$$

$$dtp(pc_1) \quad \leftarrow \tag{6}$$

to (4). Then each stable model of the rules (4–6) contains the atom $dtp(pc_1)$ and at least one of $\{part(pc_1, fscan), part(pc_1, sscan)\}$. This holds because $dtp(pc_1)$ is justified by rule (6) and, hence, a choice from $\{part(pc_1, fscan), part(pc_1, sscan)\}$ is justified by (5). If $part(pc_1, sscan)$ is taken, then by (4) the choice between $\{part(pc_1, sc_1), part(pc_1, sc_2)\}$ is justified. So this implies that $\{dtp(pc_1), part(pc_1, fscan), part(pc_1, sscan), part(pc_1, sc_1)\}$ is one stable model for the rules (4–6) but this not the case for the set $\{dtp(pc_1), part(pc_1, fscan), part(pc_1, sc_1)\}$ as $part(pc_1, sc_1)$ is not justified. We note that there is an important difference between a rule with and a rule without a head: only the former can bring something into a stable model. The latter only exclude stable models.

For applications it is very useful to provide rules where variables quantifying over the whole rule can be used. With the help of variables one can write rules in a compact and structured way and this facilitates developing, updating and maintaining a rule set.

The semantics for rules with variables [7] is obtained by considering the ground instantiation of the rules with respect to their Herbrand universe, i.e. all variable-free (ground) terms that can be built from the constants and functions in the rules. However, in this general case the rule language is highly undecidable. A decidable subclass of rules with variables is obtained by considering the function-free and *domain-restricted* case [7]. Here the intuition is that each variable has a domain predicate which provides the domain over which the variable ranges. Domain predicates can be defined using normal rules starting from basic facts. Hence, it is possible to define new domain predicates from already defined ones using union, intersection, complement, join, and projection. As an example, consider two sets of ground facts $\{ide_{hd}(h_i) \leftarrow\}$ and $\{ide_{cd}(c_i) \leftarrow\}$ giving the available IDE hard disks and IDE CD ROMs, respectively. Now both ide_{hd} and ide_{cd} can be used as domain predicates. We can also define a new domain predicate ide as their union using the two rules

$$ide(X) \quad \leftarrow ide_{hd}(X) \quad ; \quad ide(X) \quad \leftarrow ide_{cd}(X)$$

where the variable X quantifies universally over a rule. Note that we use ";" as a separator of rules. We use the convention that variables start with a capital and constants with a lower case letter. A rule is domain-restricted if every variable in it appears in a domain predicate in the body of the rule. E.g., the rule

$$1 \ \{part(X, d_1), \ldots, part(X, d_m)\} \ 2 \leftarrow pc(X) \tag{7}$$

is domain-restricted if $pc(X)$ is a domain predicate. A simple way of understanding the semantics of the rules with variables is to see them as shorthands for sets of ground rules. For example, rule (7) can be understood as a set of ground rules of form

$$1 \ \{part(pc_i, d_1), \ldots, part(pc_i, d_m)\} \ 2 \leftarrow pc(pc_i) \tag{8}$$

where X is replaced by each constant pc_i for which $pc(pc_i)$ holds.

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. Thus, rule (7) can be expressed as

$$1 \ \{part(X, D) : ide(D)\} \ 2 \leftarrow pc(X) \tag{9}$$

A rule with conditional literals is *domain-restricted* if each variable in it appears in a positive domain predicate in body or the conditional part (like $ide(D)$ above) of some conditional literal in the rule.

When using conditional literals we need to distinguish between *local* and *global* variables in a rule. The idea is that global variables quantify universally over the whole rule but the scope of a local variable is a single conditional literal. We do not introduce any notation to make the distinction explicit but use 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 to the rule. For example,

in the rule (9) the variable D is local to the conditional literal in the head but X is a global variable. Again a rule with conditional literals can be understood as a shorthand for a set of ground rules resulting from a two step process. First all global variables are replaced by all ground terms satisfying the domain predicates in every possible way. This gives a set of rules where variables appear only as local variables in conditional literals. For example, for the rule (9) elimination of the global variable X leads to a set of rules of the form

$$1\ \{part(pc_i, D) : ide(D)\}\ 2 \leftarrow pc(pc_i) \tag{10}$$

one for each ground term pc_i for which $pc(pc_i)$ holds. In the second step, for each such rule, local variables are eliminated by replacing each conditional literal by a sequence of ground instances of the main predicate such that domain predicate in the conditional part holds. For example, in rule (10) the conditional literal $part(pc_i, D)\ :\ ide(D)$ is replaced by a sequence

$$part(pc_i, d_1), \ldots, part(pc_i, d_m)$$

where the ground terms $d_1, \ldots, d_m$ are the only terms for which $ide(d_i)$ holds. Hence, the rule (9) can been seen as a shorthand for a set of ground rules of the form (8). Conjunctions of form

$$part(P, D) : pc(P) : ide(D)$$

are also allowed in conditional parts, corresponding to a sequence of ground facts $part(pc_i, d_j)$ for which both $pc(pc_i)$ and $ide(d_j)$ hold.

Domain predicates such as $cost(X, C)$ giving the unique cost C of each component X can also be used for defining weights:

$$\leftarrow 900 \leq \{part(X, Y) : cost(Y, C) = C\},\ \ pc(X) \tag{11}$$

is a shorthand for a set of ground rules containing a rule

$$\leftarrow 900 \leq \{part(pc_i, d_1) = c_1, \ldots, part(pc_i, d_n) = c_n\},\ pc(pc_i)$$

for each pc_i s.t. $pc(pc_i)$ holds and where $(d_1, c_1), \ldots, (d_n, c_n)$ are the only pairs for which the domain predicate $cost(d_i, c_i)$ holds.

Example 2. Consider Example 1. Using the rule language the configuration model can be represented in the following way. Basic component types are given as corresponding facts and normal rules defining domain predicates as for ide. The rule (9) captures the requirement that a PC has from one to two IDE devices and a rule

$$\leftarrow \{part(X, D) : ide_{hd}(D)\}\ 0,\ \ pc(X)$$

the property that at least one of them is an IDE hard disk. The requirements for the scanners and SCSI cards are captured using rules (4–5) and the constraint on the total price of the components can be stated using a rule like (11).

An implementation of the weight constraint rule language called **Smodels** is publicly available at `http://www.tcs.hut.fi/Software/smodels/`. It computes stable models of domain-restricted rules using a precompilation technique and a Davis-Putnam like procedure which employs efficient search space pruning techniques and a powerful dynamic application-independent search heuristic. The current implementation supports only non-recursive definitions of domain predicates for efficiency, and integer weights in order to avoid complications due to finite precision of standard real number arithmetic. **Smodels** is competitive even against special purpose systems, e.g., in planning and satisfiability problems [6].

3 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 new configuration model and are enclosed in a box in the following. The latter appear only in the examples. We further make the convention that the domain predicates are typeset normally whereas other predicates defining the configuration are typeset in $\boldsymbol{boldface}$.

The representation is based on a simplified version of a general configuration ontology [11]. 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. In our approach, it is represented as a set of rules. Configuration solution knowledge specifies a configuration, defined in our approach as a stable model of the set of rules in the configuration model. Requirements knowledge specifies the requirements on a configuration and is defined as a set of rules that a configuration must satisfy.

3.1 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: *component types*, *resource types*, *port types* and *constraints*. The types are organized in a *taxonomy* or *class hierarchy* where a subtype *inherits* the properties of its supertypes in the usual manner. For simplicity we only allow individuals of concrete, i.e. leaf, types of the taxonomy, which unambiguously describe the product, in a configuration.

Individuals of concrete port and 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 that are included in a configuration are represented by the unary predicate $\boldsymbol{in}()$ ranging over individuals.

The type predicates are used as the 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 type hierarchy is represented using rules stating that the individuals of the subtype are also individuals of the supertype. This effects monotonic inheritance of the property definitions.

Example 3. 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. Port and resource types are introduced in the following sections. The following rules define the component types and their taxonomy:

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

3.2 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 in which a component individual is 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 component type*. An individual of this type serves as the root of the product structure. In a configuration there is exactly one individual of the root type. Component types are also for simplicity assumed to be exclusive meaning that a component individual is part of at most one component individual. Further, no component type can have itself or any of its super- or subtypes as a possible part type in any of its part definitions or the part definitions of its possible part types, and so on recursively. This implies that a component individual is not 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 has as a part another component individual with a given part name is represented by the tertiary predicate $\boldsymbol{pa}()$ on the whole component individual, the part component individual and the part name. A part name is represented as an object constant and the set of part names in a configuration model are captured using the domain predicate $pan()$.

A part definition is represented as a rule that employs a cardinality constraint in the head. The individuals of possible part types in a given part definition of a given component type are represented using a domain predicate ppa. It is defined as the union of the individuals of the possible component types.

Example 4. 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 component type and the part names and possible part types of PC are represented as follows:

$$\begin{array}{rl;{5pt/5pt}rl} root(C) & \leftarrow pc(C) & ppa(C_1, C_2, ms) & \leftarrow pc(C_1), ide(C_2) \\ pan(ms) & \leftarrow & ppa(C_1, C_2, swp) & \leftarrow pc(C_1), sw(C_2) \\ pan(swp) & \leftarrow & & \end{array}$$

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 is in a configuration, and that other component individuals are in a configuration if they are parts of something are given as follows:

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

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:

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

3.3 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 a tertiary domain predicate $prd()$ on component individuals of the producing or using component type, individual of the produced or used resource type and the magnitude. Use is represented as negative magnitude.

Example 5. 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:

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

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:

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

3.4 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 binary domain predicate $cmb()$ on port individuals of compatible port types and a rule that any two compatible port individuals can be connected. The connections are represented as the symmetric, irreflexive binary predicate $\boldsymbol{cn}()$ on two port individuals.

A port individual is connected to at most one other port individual. The following rules represent these ontological definitions:

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

Example 6. 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 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:

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

A port definition of a component type is represented as a rule very similar to a part definition, but with the tertiary predicate po signifying that a component individual has a port individual with a given port name. The pon predicate captures the port names.

Example 7. 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:

$$
\begin{array}{lcl}
pon(ide_1) \leftarrow\ ;\ pon(ide_2) \leftarrow\ ;\ pon(ide_p) \leftarrow & & \\
1\ \{\boldsymbol{po}(C,P,ide_1) : ide_c(P)\}\ 1 & \leftarrow & \boldsymbol{in}(C), pc(C) \\
1\ \{\boldsymbol{po}(C,P,ide_2) : ide_c(P)\}\ 1 & \leftarrow & \boldsymbol{in}(C), pc(C) \\
1\ \{\boldsymbol{po}(C,P,ide_p) : ide_d(P)\}\ 1 & \leftarrow & \boldsymbol{in}(C), ide(C) \\
\leftarrow ide(C), ide_d(P_1), \boldsymbol{po}(C,P_1,ide_p),\ \{\boldsymbol{cn}(P_1,P_2) : prt(P_2)\}\ 0 & &
\end{array}
$$

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:

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

3.5 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 are used for this in our approach.

Example 8. In the PC configuration model, there is a constraint that a hard disk of type hd must be part of PC:

$$\leftarrow pc(PC_1),\ \{\boldsymbol{pa}(C_1,C_2,N) : hd(C_2) : pan(N)\}\ 0$$

4 Computational Complexity

We make the following assumptions. A configuration model $\mathcal{CM}$ represented according to the ontology is translated to a set of rules CM as in Section 3, including the rules for ontological definitions. Then, a set of ground facts S is added, providing the individuals that can be in a configuration. 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. The set S 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. However, for any configuration model $\mathcal{CM}$ agreeing with the ontology in Section 3, a finite, bounded set $max_i(\mathcal{CM})$ containing the maximum number of individuals of any concrete type can be shown to exist. It can be constructed using the following observations. For each concrete resource type there is exactly one individual. Every concrete component individual is a node in the maximal compositional structure tree rooted at the unique root component individual. The tree can be constructed by going through the part definitions starting from the root component type. Finally, the number of port individuals is bounded by the number of component individuals. Now, a set S defining enough individuals for any correct configuration w.r.t. $\mathcal{CM}$ can be constructed by adding a fact $t(i_j) \leftarrow$ for each individual i_j in $max_i(\mathcal{CM})$ whose concrete type is t.

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 c_l. 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 at most five variables, this assumption implies that there is a bound $c = \max(c_l, 5)$ on the number of variables in any rule of any CM and R.

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

Definition 9. 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 4.1. *CONFIGURATION TASK(D) is* ***NP****-complete in the size of* $CM \cup S \cup R$.

Proof. (Sketch) Task is in **NP**: For CONFIGURATION TASK(D) there is the following polynomial time non-deterministic algorithm:

1. Guess a configuration C. It holds that C is of polynomial size w.r.t. $|CM \cup S \cup R|$ since C is a subset of the Herbrand Base (H_B) of $CM \cup S$ and $|H_B|$ is at most polynomial w.r.t. $|CM \cup S|$. The latter holds since the number of variables in any rule of $CM \cup S$ is bounded by c and the size of the Herbrand Universe of CM is bounded by $|CM \cup S|$.
2. Check that C is a stable model of $CM \cup S$. This is accomplished by instantiating $CM \cup S$ with its Herbrand Universe, thus obtaining the set of ground rules $(CM \cup S)_G$, and checking that C is a stable model of $(CM \cup S)_G$. Since the number of variables in any rule of $CM \cup S$ is bounded by c and the size of the Herbrand Universe of CM is bounded by $|CM \cup S|$, $|(CM \cup S)_G|$ is polynomial w.r.t. $|CM \cup S|$ and the instantiation can also be computed in polynomial time. Checking if C is a stable model of a set of instantiated rules can be done in polynomial time [7].
3. Check that C satisfies R. This can be accomplished in polynomial time similarly as in Step 2 by instantiating $R \cup S$ with its Herbrand Universe and checking that C satisfies $(R \cup S)_G$.

NP-hardness: The **NP**-complete 3-SAT problem of whether a propositional sentence $(l_1 \vee l_2 \vee l_3) \wedge \ldots \wedge (l_{m-2} \vee l_{m-1} \vee l_m)$

is satisfiable can be reduced to the configuration task e.g. as follows: Introduce in $\mathcal{CM}$ a root component type r and add a distinct component type for each variable that appears in the sentence. Add to r a part definition for each variable such that the component type is the same as the variable, and the cardinality is $[0, 1]$. For each clause $(l_1 \vee l_2 \vee l_3)$, introduce the constraint rule $\leftarrow\ t(l_1), t(l_2), t(l_3)$, where $t(l_i) = \text{not}\ \ in(X), v_i(X)$ if l_i is a positive literal and $t(l_i) = in(X), v_i(X)$ if l_i is a negative literal, where v_i is the variable in l_i. Finally, construct S by including a fact $r(r_1) \leftarrow$ and the facts $v_i(v_{i1}) \leftarrow$ for each variable v_i. Now, the sentence is satisfiable iff there is a configuration of CM and S satisfying $R = \emptyset$.

□

5 Previous 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. Our approach differs from these in the following respects. The underlying languages do not always have a clear declarative semantics or their implementations are incomplete. They were usually not designed for configuration typical knowledge representation. Finally, the complexity of the configuration task has not been precisely classified. The last issue holds also for the approaches discussed next.

Some exceptions to the above pattern exist. In [1], a formal definition of configuration as constructing a Herbrand model of a description logic-like language is given. Another approach defining (implicitly) configurations as models of a language is described in [4], where a component and connection based ontology is formalized using Ontolingua. In [1], the ontological foundation of configuration is not explored, whereas in [4] the ontology is more restricted than ours.

The approach with perhaps the most similar goals to ours is described in [2, 3]. A similar configuration domain oriented conceptualization and a configuration model are given a formal semantics using a restricted variant of predicate logic extended with set constructs [2]. This definition is then mapped to a generative constraint satisfaction problem (GCSP), and implemented using its implementation. In our approach, the formalization is done directly using the language that provides the implementation. In contrast to our approach, the configuration task is cast as the task of finding a subset minimal set of sentences representing the configuration such that the union of the configuration model and the configuration together with requirements is consistent. The subset minimality condition is a stronger form of justification than ours based on a fixpoint definition [7].

The mapping from the conceptualization to the formalization in [2] is more complex than in our approach. This is partially due to a more complex conceptualization, but there are also other significant differences. First, unlike in [2], our mapping is modular in the sense that each type, property definition and constraint can be formalized independently of other things in the configuration model. The structure of the product is represented in [2] using ports and connections, which leads to more complexity. Furthermore, defining the configuration as a stable Herbrand model eliminates the need for "closure" axioms required in [3, 2] to restrict the set of sentences corresponding to a configuration to be complete and not to contain extraneous things. In addition, the weight constraints allow a compact representation of things which require complex predicates in [2].

It is relatively easy to extend our representation to cover a more complex conceptualization such as those in [11, 2]. These include additional concepts for representing attributes and functionality and additional definitions for the concepts. However, some aspects cannot be captured in a straightforward manner. E.g., the implementation does not yet include arithmetic for reals which is useful for expressing, e.g., attribute values and resource amounts. However, integer arithmetic is included as built-in functions.

6 Conclusions and Future Work

We have presented an approach to formally defining different conceptualizations of configuration knowledge, modeling configuration knowledge, and implementing the configuration task on the basis of the formalization. The approach is based on a novel weight constraint rule language designed for representing different aspects of configuration knowledge uniformly and in a straightforward manner. Using the language to represent different aspects of configuration knowledge is demonstrated through formalizing a unified configuration ontology. 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 implementation with these and further aspects of configuration 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

[1] M. Buchheit, R. Klein, and W. Nutt, 'Constructive problem solving: A model construction approach towards configuration', Technical Memo TM-95-01, DFKI, (1995).

[2] A. Felfernig, G. Friedrich, and D. Jannach, 'Uml as domain specific language for the construction of knowledge based configurations systems', in *Proc. of 11th Int. Conf. on Softw. Eng. and Knowl. Eng.*, (1999).

[3] G. Friedrich and M. Stumptner, 'Consistency-based configuration', in *Configuration. AAAI Technical Report WS-99-05*, pp. 35–40, (1999).

[4] T.R. Gruber and R. Olsen, 'The configuration design ontologies and the vt elevator domain theory', *Human-Computer Studies*, **44**, 569–598, (1996).

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

[6] I. Niemelä, 'Logic programming with stable model semantics as a constraint programming paradigm', *Annals of Mathematics and Artificial Intelligence*, **25**, 241–273, (1999).

[7] I. Niemelä, P. Simons, and T. Soininen, 'Stable model semantics of weight constraint rules', in *Proc. of the Fifth Intern. Conf. on Logic Programming and Nonmonotonic Reasoning*, pp. 317–331, (1999).

[8] P. Simons, 'Extending the stable model semantics with more expressive rules', in *Proc. of the Fifth Intern. Conf. on Logic Programming and Nonmonotonic Reasoning*, pp. 305–316, (1999).

[9] T. Soininen, E. Gelle, and I. Niemelä, 'A fixpoint definition of dynamic constraint satisfaction', in *Proc. of the 5th International Conf. on Principles and Practice of Constraint Programming*, pp. 419–433, (1999).

[10] T. Soininen and I. Niemelä, 'Developing a declarative rule language for applications in product configuration', in *Proc. of the First Int. Workshop on Practical Aspects of Declarative Languages*, pp. 305–319, (1999).

[11] T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a general ontology of configuration', *AI EDAM*, **12**, 357–372, (1998).