**Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH**

**Technical
Memo**

TM-95-01

Constructive Problem Solving: A Model Construction Approach towards Configuration

Martin Buchheit, Rüdiger Klein, Werner Nutt

Januar 1995

Deutsches Forschungszentrum für Künstliche Intelligenz GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

Deutsches Forschungszentrum für Künstliche Intelligenz

The German Research Center for Artificial Intelligence (Deutsches Forschungszentrum für Künstliche Intelligenz, DFKI) with sites in Kaiserslautern and Saarbrücken is a non-profit organization which was founded in 1988. The shareholder companies are Atlas Elektronik, Daimler-Benz, Fraunhofer Gesellschaft, GMD, IBM, Insiders, Mannesmann-Kienzle, Sema Group, Siemens and Siemens-Nixdorf. Research projects conducted at the DFKI are funded by the German Ministry for Research and Technology, by the shareholder companies, or by other industrial contracts.

The DFKI conducts application-oriented basic research in the field of artificial intelligence and other related subfields of computer science. The overall goal is to construct systems with technical knowledge and common sense which - by using AI methods - implement a problem solution for a selected application area. Currently, there are the following research areas at the DFKI:

- ☐ Intelligent Engineering Systems
- ☐ Intelligent User Interfaces
- ☐ Computer Linguistics
- ☐ Programming Systems
- ☐ Deduction and Multiagent Systems
- ☐ Document Analysis and Office Automation.

The DFKI strives at making its research results available to the scientific community. There exist many contacts to domestic and foreign research institutions, both in academy and industry. The DFKI hosts technology transfer workshops for shareholders and other interested groups in order to inform about the current state of research.

From its beginning, the DFKI has provided an attractive working environment for AI researchers from Germany and from all over the world. The goal is to have a staff of about 100 researchers at the end of the building-up phase.

Dr. Dr. D. Ruland

Director

Constructive Problem Solving:
A Model Construction Approach towards Configuration

Martin Buchheit, Rüdiger Klein, Werner Nutt

DFKI-TM-95-01

A version of this paper appeared in the Proceedings of the 3rd International Conference on Artificial Intelligence in Design, AID'94, Lausanne, Switzerland.

This work has been supported by a grant from The Federal Ministry for Research and Technology (FKZ ITWM-9201).

ISSN 0946-0071

Constructive Problem Solving: A Model Construction Approach towards Configuration

Martin Buchheit, Werner Nutt
German Research Center for
Artificial Intelligence - DFKI GmbH
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
e-mail: {buchheit, nutt}@dfki.uni-sb.de
tel. (+49 681) 302 {52 57, 53 25}

Rüdiger Klein
Daimler-Benz AG
AI Research Group
Postfach 48 01 04
12251 Berlin, Germany
e-mail: klein@DBresearch-berlin.de
tel. (+49 30) 399 82 211

Abstract

In this paper we give a formalisation of configuration as the task to construct for a given specification, which is understood as a finite set of logical formulas, a model that satisfies the specification. In this approach, a specification consists of two parts. One part describes the domain, the possible components, and their interdependencies. The other part specifies the particular object that is to be configured. The language that is used to represent knowledge about configuration problems integrates three sublanguages that allow one to express constraints, to build up taxonomies, and to define rules.

We give a sound calculus by which one can compute solutions to configuration problems if they exist and that allows one to recognize that a specification is inconsistent. In particular, the calculus can be used in order to check whether a given configuration satisfies the specification.

Contents

1 Introduction

In recent years configuration problem solving has become an established field of application of knowledge-based systems. There are mainly two aspects which contributed to this development. On the one hand the great potential of application: this results in a growing interest in knowledge-based techniques by customers who expect significant support in solving their configuration problems. On the other hand, a main characteristic of configuration problems is their well-structuredness, which provides much better opportunities for an adequate description with AI techniques available today than there are in many other fields.

Nevertheless, we are still lacking a *comprehensive theory* of configuration problem solving. There are some approaches, emphasizing this or another aspect, and there is a certain number of implemented systems [Hei88, CGS90, CG91]. The main problem with these approaches and systems seems to be the missing "global" problem solving paradigm.

Configuration problem solving is basically a synthetic or constructive activity [Kle91]. The solution to a given problem has to be synthesized in such a way that it fulfils the problem specification and that it is in accordance with certain constraints. The synthesis of a solution includes different problem solving activities: object selection and creation ("instantiation"), specialization, parametrization, hierachical refinement, etc. These operations have to be performed in such a way that they are consistent with the constraints governing the application domain.

The resulting knowledge representation and problem solving can be very complex. Up to now many application-oriented AI configuration systems have been developed that tried to meet these demands with more or less ad-hoc techniques (and in some cases are used in practice with considerable success). But sooner or later—due to the missing global problem solving paradigm—such systems run into difficulties comparable to those encountered in traditional software applications: problems of maintenance, of extendability, of re-usability etc.

Compared with traditional software techniques, the main advantages of knowledge-based approaches are in declarative representation languages together with inference rules that reflect the declarative semantics [Kow90, Llo90]. Configuration problems seem to be well-suited for the development of declarative problem solving techniques for the following reasons:

- Since the objects involved are technical in nature, their properties are well-defined, and their relationships are governed by precise constraints. In problem specifications, implicitness, ambiguities, and vagueness play a much less prominent role than in many other fields.
- Configuration problems do not include any notion of belief, of time, or of changing worlds.
- The meta-assumptions underlying a problem specification can often be represented explicitly.

Other problem classes of synthetic or constructive character, like design, planning, speech synthesis, explanation generation, etc., are complicated by the presence of vague or un-

certain information. Thus, many aspects which in other areas make formally well-founded knowledge representation and inferencing techniques complicated are not present in configuration problems. However, the demands for knowledge representation, problem solving, and control in this problem class still make it a challenging task to develop a comprehensive, adequate, and formally well-founded approach.

The goal of this paper is to give a formal description of a declarative problem solving paradigm that integrates techniques which have proved to be essential for solving configuration tasks. Three kinds of formalisms can be distinguished that seem to be essential for representing knowledge about configuration problems.

First, one needs a language that allows one to represent taxonomies. Families of technical devices are usually described as hierarchies, where the concrete objects show up at the minimal positions, and also systems to be configured can be represented by a taxonomy. Representing knowledge about devices and systems in this way allows one to build up a configuration by stepwise refinement, pushing an object through the hierarchy until it reaches a minimal position. In addition, one can use inheritance in order to represent properties of objects in a structured way. Due to the nature of the domain, inheritance is monotonic.

Languages of the KL-ONE family (also called concept languages or description logics) have been designed as a tool for representing knowledge and to reason about taxonomies (see $\mathcal{KRIS}$ [BH91c], CLASSIC [BBMR89], KRIPTON [BPGL85], LOOM [MB87], BACK [QK90]). These languages have been given a formal semantics that identifies them as a fragment of first order predicate logic. Based on this semantics, recent research in algorithms for such languages and in the complexity of the inferences they perform has led to a thorough understanding of this kind of formalism (see [SSS91, DLNN91a, DLNN91b]).

Second, languages to express constraints and to propagate them are indispensable. In AI, constraint propagation and satisfaction is an active research area and a substantial body of results are available ([Van89, Mac87]).

Third, relations and rules are necessary in order to describe how a specification of the function of a system can be transformed into a structural description. Moreover, rules can describe how abstract relationships between objects can be refined to more concrete ones, that can be technically realized.

Several attempts have been made to integrate the above formalisms. For instance, constraint logic programming combines constraints and rules ([JL87, HS88]). Recently, Baader and Hanschke have developed a schema for the integration of constraint languages and KL-ONE like taxonomic languages, provided the constraint language satisfies certain requirements ([BH91a]). For clausal logic and several restricted forms of taxonomies calculi have been designed that are based on the idea of order-sorted unification ([Wal87, SS89]).

We will apply these ideas to our framework as guidelines to the integration of its components. In particular, we will use the Baader/Hanschke approach for the integration of taxonomies and constraints. Taxonomies and constraints together will form a "constraint language" in the sense of Höhfeld/Smolka, for which there is a canonical way to extend it by rules to a constraint logic programming language.

As mentioned above, configuration problems are of a constructive nature and solutions are descriptions of technical systems. We will capture this constructive aspect by formalizing

a configuration problem as the task to generate a model for a given specification. Thus, our approach to configuration is not a deductive one but can rather be understood as an abductive one.

The rest of the paper is organized as follows. In Section 2 we present in more detail the basic idea to view configuration as a model generation problem and introduce the different components a problem specification consists of. In Section 3 we define the three kinds of knowledge representation languages that we are going to use, namely languages for constraints, for taxonomic descriptions, and for relations and rules. In Section 4 we say how these languages are used on the one hand to express definitional knowledge, integrity constraints, and specifications of systems, and on the other hand to describe models. In Section 5 we give a calculus that computes a model for a given problem specification. Section 6 concludes.

2 Configuration as Model Construction

We will now outline the basic idea underlying our formalization of configuration. In this and the following sections we assume that the reader is familiar with some logical and model theoretical background as provided by textbooks on mathematical logics, e.g. [Sho67].

The basic idea can be described as follows: a configuration problem solver gets as input a set of logical formulas that specifiy the system to be configured and produces as output a model of this specification. Of course, this idea needs some refinement in order to be useful.

We suppose that we are given a logical language, with signature Σ, that allows us to talk about the systems we are interested in. In addition we assume that there is a sublanguage, called the *basic* language,—whose signature Σ_0 is a subsignature of Σ—that gives us the vocabulary to name the technical devices by which a concrete system is built up as well as their relationships. For instance, one can imagine that the names of the technical devices as they show up in a catalogue are part of Σ_0. We assume that the basic language is rich enough to describe concrete systems. We distinguish between the two languages because the specification of a system to be configured will be given using the overall language, whereas descriptions of models that satisfy the specification consist only of basic formulas.

The representation of knowledge about technical systems will consist of two parts. First, there will be *definitional* knowledge, that relates the abstract notions, i.e., the elements of $\Sigma \setminus \Sigma_0$, to the basic language. The definitional knowledge will be represented by a set $\mathbf{D}$ consisting of taxonomic hierarchies and rules. By means of $\mathbf{D}$ one can extend every Σ_0-structure C to a Σ-structure $\tilde{C}$ in a unique way. Second, there will be a set $\mathbf{I}$ of *integrity constraints* that express necessary conditions which the components of a configuration have to satisfy or rule out certain combinations of components as impossible. We call $\mathbf{D}$ and $\mathbf{I}$ together *domain knowledge*. The *specification* of a system to be configured will be given as a finite set $\mathbf{S}$ of formulas of the general language. In order to define more precisely which structures are solutions to such a specification we need some technical definitions.

Let C be a Σ_0-structure. If F is a set of Σ-formulas we write

$$C \models_{\mathbf{D}} F$$

in order to express that F holds in the extension $\tilde{C}$ of C, which is defined by $\mathbf{D}$. Intuitively,

this means that F holds in C if we use the definitional knowledge $\mathbf{D}$ to translate arbitrary Σ-expressions into the basic language.

Now suppose that $\mathbf{D}$ and $\mathbf{I}$ are given and that $\mathbf{S}$ is a specification. Then a *configuration* $\mathbf{C}$ is a finite Σ_0-structure such that the following holds:

- $\mathbf{C} \models_{\mathbf{D}} \exists.\mathbf{S}$, where $\exists.\mathbf{S}$ is the existential closure of $\mathbf{S}$
- $\mathbf{C} \models_{\mathbf{D}} \mathbf{I}$.

The two conditions simply say that $\mathbf{C}$ has to satisfy on the one hand the specific requirements expressed by $\mathbf{S}$ and on the other hand the general requirements that are imposed on all systems.

Following this schema, we will formally introduce in the next chapters the languages that we want to use for describing configuration problems.

3 The Knowledge Representation Languages

As mentioned in the Introduction, the language that we want to use for the description of configuration problems consists of three components: one for expressing constraints that can be propagated and solved by special purpose algorithms, another one by which we can describe classes with attributes that are organized as a taxonomy, and finally a general language for relations and rules.

In our approach we will assume that constraints are handled by a kind of black box. An appropriate abstraction of such a component is to conceive it as a "concrete domain" as defined by Baader and Hanschke [BH91b]. In 3.1 we briefly review this concept and discuss which aspects of an application can be covered by it.

Next we devise a language that allows one to describe classes of objects using constructs that are known from KL-ONE-like description logics. This language incorporates concrete domains as proposed in [BH91b]. It can be seen as an extension of the concrete domain language to a more general constraint language.

Finally, we extend the general constraint language by relations along the lines of the constraint logic programming scheme described in [HS88].

General Assumption. *All languages share one set of variables and one set of constants. There are no terms other than variables or constants. Variables are denoted by the letters x, y, z, constants by a, b, c, and terms by s, t, u.*

3.1 Concrete Domains

The notion "concrete domain" has been introduced by Baader and Hanschke in [BH91b] in the context of KL-ONE like concept languages. A drawback of such languages is that all terminological knowledge has to be defined on an abstract logical level. In many applications, however, one would like to refer to fixed domains and predicates when defining concepts. Examples of such domains are the real numbers, integers, finite domains that

are explicitly given by an enumeration of their elements etc. The fixed predicates could be equality, inequality, extensionally defined predicates or other more complex predicates. Especially in the context of configuration one needs the integers or the reals in order to express relations between "concrete valued" features.

Technically, one can see a concrete domain as a black box. The input-output behaviour of this black box will be described below. The following definition formalizes the notion "concrete domain" (see [BH91b]).

A *concrete domain* $\mathcal{D}$ consists of a set $dom(\mathcal{D})$, the domain of $\mathcal{D}$, and a set $pred(\mathcal{D})$, the predicate names of $\mathcal{D}$. Each predicate name P is associated with an arity n and an n-ary relation $P^{\mathcal{D}} \subseteq dom(\mathcal{D})^n$. Each element of $dom(\mathcal{D})$ is named by a unique constant. Such constants are called *concrete constants*.

As an example, the integers or reals form concrete domains $\mathbf{Z}$ or $\mathbf{R}$, respectively, if we take as predicates all formulas which are built by first order means from equalities and inequalities between polynomials like, for instance, $\exists y.(2x + y > z^2 \wedge y < x)$.

If we want to combine inference algorithms for concept languages with reasoning algorithms for the concrete domain then the concrete domain must satisfy some additional properties.

A set of predicate names of a concrete domain $\mathcal{D}$ is *closed under negation*, if for each predicate P of arity n in $pred(\mathcal{D})$ there exists some $\bar{P}$ in $pred(\mathcal{D})$ with the same arity such that $\bar{P}^{\mathcal{D}} = dom(\mathcal{D})^n \setminus P^{\mathcal{D}}$.

Let $P_1, \ldots, P_m$ be m (not necessarily different) predicate names in $pred(\mathcal{D})$ of arities $n_1, \ldots, n_m$. We consider the conjunction

$$\bigwedge_{l=1}^{m} P_l(\vec{t}^{(l)}).$$

Here $\vec{t}^{(l)}$ stands for an n_l-tuple $(t_1^{(l)}, \ldots, t_{n_l}^{(l)})$ where each $t_i^{(l)}$ is either a variable or a concrete constant. Note that neither the terms in one tuple nor those in different tuples are assumed to be distinct. Such a conjunction is said to be *satisfiable* iff there exists an assignment that maps each variable to an element of $dom(\mathcal{D})$ such that the conjunction becomes true in $\mathcal{D}$.

A concrete domain $\mathcal{D}$ is called *admissible* iff (i) the set of its predicate names is closed under negation, and (ii) the satisfiability problem for finite conjunctions of the above mentioned form is decidable.

Reconsidering our previous examples $\mathbf{Z}$ and $\mathbf{R}$, we find that in both cases the set of predicates is closed under negation, but $\mathbf{Z}$ is not admissible because of the undecidability of Hilbert's Tenth Problem [Mat70], whereas $\mathbf{R}$ is admissible because of Tarski's decidability result for real arithmetic. If one transforms $\mathbf{Z}$ into $\mathbf{Z}_{\mathcal{P}}$ which is obtained by taking as predicates only formulas that have been built up using linear polynomials, the resulting concrete domain is again admissible, since Presburger Arithmetic is decidable.

The examples that we have given up to now suggest that a concrete domain has to be somewhat homogeneous in nature, like the integers or reals, but this need not necessarily be the case. The domain can consist, e.g., of the union of all integers with the booleans and with an enumeration of colours like $\{red,\ blue,\ green\}$, and there may be predicates that relate numbers, booleans, and colours in various fashions. In particular, the notion of admissible concrete domains allows one to capture collections of several constraint solvers.

3.2 Descriptions and Taxonomies

Our aim in this subsection is to define a description language that is based on sorts, features, and concrete domains. We will use this language to build up taxonomies by which we describe the sorts of the objects that our domain consists of. The language we propose takes up ideas that have emerged on the one hand from the research in KL-ONE and its descendants and on the other hand from the development of feature descriptions that are used in computational linguistics. The main characteristic of these languages is that descriptions of classes of objects are given in terms of superclasses and attributes[1]. The main inferences for descriptions are to decide whether a description is consistent—i.e., satisfiable—and whether one description is more general than another one—i.e., whether it subsumes the other.

The language we propose is such that it combines maximal expressivity with a complexity as low as possible. In particular, it is chosen in such a way that the above mentioned inferences are decidable. Moreover, we have to take into account requirements that are due to the structure of our application. It seems that in order to give sensible descriptions of technical objects one needs attributes that take values in concrete domains and has to be able to say that chains of attributes have different or identical fillers.

Throughout this subsection we assume that $\mathcal{D}$ is a concrete domain with set of predicates $pred(\mathcal{D})$.

3.2.1 Descriptions with Sorts and Features

We assume that in addition to the symbols in $pred(\mathcal{D})$ two more sets of symbols are given: a set of *sorts* (denoted by the letter S) and a set of *features* (denoted by the letters f, g).

A *path* (denoted by the letters p, q) is a—possibly empty—sequence $f_1 \cdots f_n$ of features. The empty path is denoted by ε.

Using concrete predicates, sorts, and features as building blocks, we form descriptions D, D' according to the following syntax rules:

$$
\begin{array}{lcll}
D, D' & \longrightarrow & \top \mid & \text{(top sort)} \\
 & & \bot \mid & \text{(bottom sort)} \\
 & & S \mid & \text{(sort)} \\
 & & D \sqcap D' \mid & \text{(intersubsection)} \\
 & & D \sqcup D' \mid & \text{(union)} \\
 & & \neg D \mid & \text{(complement)} \\
 & & p{\uparrow} \mid & \text{(undefinedness)} \\
 & & f.D \mid & \text{(selection)} \\
 & & p \doteq q \mid & \text{(agreement)} \\
 & & p \not\doteq q \mid & \text{(disagreement)} \\
 & & P(p_1, \ldots, p_k) & \text{(concrete predicate).}
\end{array}
$$

Next, we give a semantics for these expressions in the usual way. An interpretation is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a nonempty set that is disjoint from $dom(\mathcal{D})$ and $\cdot^{\mathcal{I}}$ is a function that maps sorts to subsets of $\Delta^{\mathcal{I}}$ and features and paths to partial functions from

[1] Attributes are called "roles" in the KL-ONE context and "features" in linguistics.

$\Delta^{\mathcal{I}}$ to $\Delta^{\mathcal{I}} \cup dom(\mathcal{D})$ such that the following equations are satisfied (by *dom* we also denote the domain of partial functions):

$$\begin{aligned}
\varepsilon^{\mathcal{I}}(a) &= a \text{ for every } a \in \Delta^{\mathcal{I}}, \text{ i.e. } \varepsilon^{\mathcal{I}} \text{ is the identity function on } \Delta^{\mathcal{I}} \\
(pf)^{\mathcal{I}}(a) &= f^{\mathcal{I}}(p^{\mathcal{I}}(a)) \text{ for every } a \in \Delta^{\mathcal{I}} \\
\top^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} &= \emptyset \\
(D \sqcap D')^{\mathcal{I}} &= D^{\mathcal{I}} \cap D'^{\mathcal{I}} \\
(D \sqcup D')^{\mathcal{I}} &= D^{\mathcal{I}} \cup D'^{\mathcal{I}} \\
(\neg D)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus D^{\mathcal{I}} \\
(p\uparrow)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \setminus dom\, p^{\mathcal{I}} \\
(f.D)^{\mathcal{I}} &= \{a \in dom\, f^{\mathcal{I}} \mid f^{\mathcal{I}}(a) \in S^{\mathcal{I}} D\} \\
(p \doteq q)^{\mathcal{I}} &= \{a \in dom\, p^{\mathcal{I}} \cap dom\, q^{\mathcal{I}} \mid p^{\mathcal{I}}(a) = q^{\mathcal{I}}(a)\} \\
(p \not\doteq q)^{\mathcal{I}} &= \{a \in dom\, p^{\mathcal{I}} \cap dom\, q^{\mathcal{I}} \mid p^{\mathcal{I}}(a) \neq q^{\mathcal{I}}(a)\} \\
(P(p_1, \ldots, p_k))^{\mathcal{I}} &= \{a \in dom\, p_1^{\mathcal{I}} \cap \cdots \cap dom\, p_k^{\mathcal{I}} \mid (p_1^{\mathcal{I}}(a), \ldots, p_k^{\mathcal{I}}(a)) \in P^{\mathcal{D}}\}.
\end{aligned}$$

For computations with sort expressions it is often convenient to assume that they are in a certain normal form. It is easy to see that every expression of our language is equivalent to an expression where the complement sign shows up only in front of sort symbols. Such an expression is said to be in *negation normal form*. One readily verifies, that every expressions can be rewritten in linear time to an expression in negation normal form (see also [SSS91, DLNN91a]).

3.2.2 Constraint Systems

We augment our description language in such a way that we can make statements about individuals. A *constraint* is a piece of syntax of one of the following forms:

$$s\colon D,\ spt,\ s \doteq t,\ s \not\doteq t,\ P(s_1, \ldots, s_k).$$

Let $\mathcal{I}$ be an interpretation. From now on we assume also that $\mathcal{I}$ maps constants to elements of $\Delta^{\mathcal{I}} \cup dom(\mathcal{D})$ in such a way that names of elements of $dom(\mathcal{D})$ are mapped to the corresponding elements and that all other constants are mapped to elements of $\Delta^{\mathcal{I}}$ in such a way that $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ for distinct constants a, b. An *$\mathcal{I}$-assignment* is a function α that maps a constant a to $a^{\mathcal{I}}$ and a variable to an element of $\Delta^{\mathcal{I}} \cup dom(\mathcal{D})$. We say that α *satisfies*

$s\colon D$, if $\alpha(s) \in D^{\mathcal{I}}$

spt, if $\alpha(s) \in dom\, p^{\mathcal{I}}$ and $p^{\mathcal{I}}(\alpha(s)) = \alpha(t)$

$s \doteq t$, if $\alpha(s) = \alpha(t)$

$s \not\doteq t$, if $\alpha(s) \neq \alpha(t)$

$P(s_1, \ldots, s_k)$, if $(\alpha(s_1), \ldots, \alpha(s_k)) \in P^{\mathcal{D}}$.

A *constraint system* is a nonempty finite sets of constraints. An $\mathcal{I}$-assignment α *satisfies* a constraint system C if α satisfies every constraint in C. A constraint system C is *satisfiable* if there is an interpretation $\mathcal{I}$ and an $\mathcal{I}$-assignment α such that α satisfies C.

Proposition 3.1 (Decidability)

1. *It is decidable whether a constraint system is satisfiable;*
2. *Independently of the particular concrete domain, satisfiability of constraint systems is an NP-hard problem;*
3. *Satisfiability of constraint systems not involving concrete predicates is NP-complete.*

Proof. The claims follow from results in [Smo88], [HN90], and [BH91b]. □

3.2.3 Descriptions and Taxonomies

As said before, we want to use our description logic to describe taxonomies. We will do so by augmenting the language by axioms that allow us to specify inclusion of sorts. Our taxonomies will have the property that every sort is the union of its subsorts. This is a property of technical domains where we have complete information about the existing objects and the classes they form. In addition, sorts will be described by necessary conditions that their elements have to satisfy, and sorts can be declared to be disjoint.

In order to express this kind of statement, we provide three kindes of axioms. A *cover axiom* has the form

$$S_0 \doteq S_1 \sqcup \cdots \sqcup S_n.$$

An *inclusion axiom* has the form

$$S \sqsubseteq D.$$

A *disjointness axiom* has the form

$$S_1 \parallel S_2.$$

Let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be an interpretation. The Interpretation $\mathcal{I}$ satisfies the cover axiom $S_0 \doteq S_1 \sqcup \cdots \sqcup S_n$ if $S_0^{\mathcal{I}} = S_1^{\mathcal{I}} \cup \cdots \cup S_n^{\mathcal{I}}$. We say that $\mathcal{I}$ satisfies the inclusion axiom $S \sqsubseteq D$ if $S^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. We say that $\mathcal{I}$ satisfies the disjointness axiom $S_1 \parallel S_2$ if $S_1^{\mathcal{I}} \cap S_2^{\mathcal{I}} = \emptyset$. An interpretation is a *model* of a set of axioms $\mathcal{A}$ if it satisfies every axiom in $\mathcal{A}$.

Let $\mathcal{A}$ be a set of axioms. A sort S *directly depends* on a sort S' if $\mathcal{A}$ contains an axiom of the form $S \doteq D$ or $S \sqsubseteq D$ such that S' occurs in D. We say that S *depends* on S' if S is related to S' by the transitive closure of the relation "directly depends." A set of axioms is *cycle free* if no sort depends on itself.

Let $\mathcal{A}$ be a finite set of axioms that is cycle free and contains for every sort S at most one cover axiom $S \doteq D$. We transform $\mathcal{A}$ into a set $\mathcal{A}^*$ by repeatedly performing the following operation. We take an inclusion $S \sqsubseteq D$ from $\mathcal{A}$ such that a sort S_0 occurs in D for which there is a cover axiom $S_0 \doteq D_0$ and replace the occurrence of S_0 in D by D_0. Since $\mathcal{A}$ is cycle free, this process eventually halts and we end up with a new set of axioms which we call $\mathcal{A}^*$.

A finite set of axioms $\mathcal{A}$ is *admissible* if the following holds:

1. for every sort S there is at most one cover axiom $S \doteq D$ in $\mathcal{A}$
2. $\mathcal{A}$ is cycle free
3. $\mathcal{A}^*$ is cycle free.

Without loss of generality we can assume that an admissible set of axioms contains for every sort S at most one inclusion of the form $S \sqsubseteq D$, since we can combine two inclusions $S \sqsubseteq D$ and $S \sqsubseteq D'$ into one axiom $S \sqsubseteq D \sqcap D'$ without changing the semantics.

Proposition 3.2 *Every admissible set of axioms has a model.*

It might be the case that some sorts are interpreted as the empty set in every model of an admissible set of axioms.

Next, we turn to the satisfiability of constraint systems w.r.t. to sets of axioms. Let $\mathcal{A}$ be a set of axioms and C be a constraint system. We say that C is *satisfiable w.r.t.* $\mathcal{A}$ if there is a model $\mathcal{I}$ of $\mathcal{A}$ and an $\mathcal{I}$-assignment α such that α satisfies C.

A sort S is *basic* if there is no cover axiom of the form $S \doteq D$. Satisfiability w.r.t. admissible sets of axioms can be reduced to satisfiability of constraint systems alone. This can be seen as follows. Let $\mathcal{A}$ be admissible. Thus we can transform $\mathcal{A}$ into a set $\mathcal{A}^*$ as described before. Note that in $\mathcal{A}^*$ the only sort symbols occurring in the right hand side of inclusions are basic symbols. Then we perform the following steps:

1. Eliminate the disjointness axioms by introducing dummy sorts (see e.g. [Neb89]).
2. Expand left hand sides of inclusions by means of the cover axioms; this leads to inclusion axioms of the form $S_1 \sqcup \cdots \sqcup S_n \sqsubseteq D$, where $S_1, \ldots, S_n$ are basic sorts.
3. Now, each such inclusion axiom is equivalent to the inclusions $S_i \sqsubseteq D$ for $i = 1, \ldots, n$; we therefore replace the former by the latter and end up with inclusion axioms only for basic sorts.
4. Transform inclusion axioms into equalities using the technique of dummy sorts (see e.g. [Neb89]).
5. At this stage, we can expand the cover axioms, using the cover axioms themselves and using the basic axioms in order to replace basic sorts.
6. We call the resulting set of axioms $\mathcal{A}^{**}$. In $\mathcal{A}^{**}$, there is exactly one definition for every sort, and no defined sort occurs in the right hand side of a definition. We call $\mathcal{A}^{**}$ the *expansion* of $\mathcal{A}$.

Let $\mathcal{A}$ be an admissible set of axioms and C be a constraint system. We transform C into a system C^{**} by replacing every occurence of a sort with its definition in $\mathcal{A}^{**}$—if it has one. We call C^{**} the $\mathcal{A}$-expansion of C.

Proposition 3.3 *Let $\mathcal{A}$ be an admissible set of axioms and C be a constraint system. Then C is satisfiable w.r.t. $\mathcal{A}$ if and only if C^{**} is satisfiable.*

Moreover, if $\mathcal{I}$ is an interpretation and α an $\mathcal{I}$-assignment satisfying C^{**}, then α satisfies C as well. Conversely, if $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is an interpretation and α an $\mathcal{I}$-assignment satisfying C, then there exists an interpretation $\mathcal{I}' = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}'})$ with the same carrier such that $A^{\mathcal{I}} = A^{\mathcal{I}'}$ for the sorts occurring in C and such that α, considered as an $\mathcal{I}'$-assignment, satisfies C^{**}. Intuitively this means that C^{**} alone has the same models as C and $\mathcal{A}$ together.

Proposition 3.4 *There exists an algorithm that decides for every admissible set of axioms $\mathcal{A}$ and every constraint system C whether C is satisfiable w.r.t. $\mathcal{A}$.*

Proof. The algorithm proceeds by first transforming C into C^{**} and then checking C^{**} for satisfiability. Such a check can be made by a calculus based on propagation rules like in [HN90] or [BH91b]. □

3.2.4 Decidability and Complexity of Satisfiability

We conclude this section by considering the satisfiability problem for classes of sets of axioms that are either less or more restrictive than admissible sets. Our results for such variants show on the one hand that slight changes may easily cause undecidability and on the other hand that it is hard to think of restrictions that reduce the complexity of reasoning in a formalism of this kind.

We say that a description is *simple* if it is built up according to the following syntax rules:

$$D, D' \quad \longrightarrow \quad \top \mid S \mid D \sqcap D' \mid f.S \mid p \doteq q \mid p \not\doteq q.$$

We say that a set of axioms is *simple* if for every inclusion $S \sqsubseteq D$ the description D is simple.

Proposition 3.5 *Satisfiability of constraint systems w.r.t. simple sets of axioms, that may contain cycles, is undecidable.*

Proof. We can reduce the word problem for semi-groups, which is known to be undecidable, to the satisfiability problem for constraints w.r.t. simple sets of axioms. Suppose $f_1, \ldots, f_m$ is a finite alphabet and p, q, $p_1, \ldots, p_n$, $q_1, \ldots, q_n$ are words over this alphabet. We view $f_1, \ldots, f_m$ also as features and p, q, $p_1, \ldots, p_n$, $q_1, \ldots, q_n$ as paths. Consider the set $\mathcal{A}$ containing the single axiom

$$S \sqsubseteq f_1.S \sqcap \cdots \sqcap f_m.S \sqcap p_1 \doteq q_1 \sqcap \cdots \sqcap p_n \doteq q_n.$$

Then the word identity $p \doteq q$ follows from the identities $p_1 \doteq q_1, \ldots, p_n \doteq q_n$ if and only if the constraint $x\colon S \sqcap p \not\doteq q$ is unsatisfiable w.r.t. $\mathcal{A}$. □

This result shows that cycle freeness is a crucial condition in order to guarantee decidability of the satisfiability problem. Intuitively, the language construct that causes undecidability is the agreement of feature paths.

Proposition 3.6 *Satisfiability of constraint systems w.r.t. simple admissible sets of axioms is NP-hard, even if agreements and disagreements are disallowed.*

Proof. We can reduce 3SAT, that is the satisfiablity problem for sets of propositional clauses with three literals, which is known to be NP-complete, to the satisfiability problem for constraint systems w.r.t. simple admissible sets of axioms. Let $u_1, \ldots, u_m$ be propositional variables and let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be a set of clauses over these variables that have three literals each. Thus, each clause has the form $C_j = l_1^{(j)} \vee l_2^{(j)} \vee l_3^{(j)}$, where $l_k^{(j)}$ is either a variable or a negated variable. We construct a set of axioms $\mathcal{A}$ as follows. Let $S_0, \ldots, S_n$, S^+, S^-, $T_1^+, \ldots, T_m^+$, and $T_1^-, \ldots, T_m^-$ be sorts, and $f_1, \ldots, f_m$ be features. Let $\mathcal{A}$ contain the following axioms:

- $S^+ \parallel S^-$
- $T_i^+ \sqsubseteq f_i.S^+$ and $T_i^- \sqsubseteq f_i.S^-$ for $i = 1, \ldots, m$
- $S_j \doteq T_1^{(j)} \sqcup T_2^{(j)} \sqcup T_3^{(j)}$ for $j = 1, \ldots, n$, where $T_k^{(j)} = T_k^+$ if $l_k^{(j)} = u_k$ and $T_k^{(j)} = T_k^-$ if $l_k^{(j)} = \neg u_k$.
- $S_0 \sqsubseteq S_1 \sqcap \cdots \sqcap S_n$

The idea underlying this construction is that S^+ and S^- encode `true` and `false`, T_i^+ and T_i^- encode the literals u_i and $\neg u_i$, S_i encodes the clause C_i, and S_0 encodes the whole set of clauses. Then it is easy to see that $\mathcal{A}$ is simple and admissible. The reader readily verifies that $\mathcal{C}$ is satisfiable if and only if the constraint $x\colon S_0$ is satisfiable w.r.t. $\mathcal{A}$. □

In this case, NP-hardness is due to the interplay of conjunctive expressions in the right hand side of inclusions and disjunctive expressions in the right hand side of cover axioms. Proposition 3.1 shows that the complexity of the satisfiability problem is not increased by allowing for a richer sort language.

One might argue that for certain applications it is not sufficient to give only necessary conditions for an individual to be an element of a sort, as it is done in sort inclusions. One might therefore want to extend the language of axioms by allowing also for sort definitions of the form

$$S \doteq D.$$

Such an extension again gives rise to undecidability.

Proposition 3.7 *Satisfiability of constraint systems w.r.t. simple cycle free sets of axioms with sort definitions is undecidable.*

Proof. Sort definitions together with cover axioms allow one again to express cyclic sets of axioms. □

3.3 The Relational Language

In this section we want to augment our description language with *relations*. This will be done by a two step construction, which is similar to that of Höhfeld and Smolka [HS88] or of Jaffar and Lassez [JL87]. In the first step we introduce a *constraint language* $\mathcal{L}$ which is based on the taxonomical language defined in section 3.2. In the second step $\mathcal{L}$ will

be extended to a *relational language* $\mathcal{R}(\mathcal{L})$ providing for relational atoms, propositional connectives and quantification.

The constraint language $\mathcal{L}$ consists of *constraint formulas*. A constraint formula is a conjunction $c_1 \;\&\; \ldots \;\&\; c_n$ of constraints $c_1, \ldots, c_n$.

An $\mathcal{I}$-assignment α *satisfies* a constraint formula $c_1 \;\&\; \ldots \;\&\; c_n$ if it satisfies all its constraints c_i. We call a constraint formula ϕ *valid in an interpretation* $\mathcal{I}$, if α satisfies ϕ for all $\mathcal{I}$-assignments α. $\mathcal{I}$ is called a *model* of a set of constraint formulas Φ, if every $\phi \in \Phi$ is valid in $\mathcal{I}$.

A constraint formula $c_1 \;\&\; \ldots \;\&\; c_n$ is *simple* if all its constraints c_i are simple. A constraint is simple if it is of the form:

$$s{:}\,D,\;\; spt,\;\; s \doteq t,\;\; s \not\doteq t,\;\; P(s_1, \ldots, s_k),$$

where D is a description according to the following syntax rule (cf. 3.2):

$$D, D' \;\longrightarrow\; \top \mid S \mid D \sqcap D' \mid f.S \mid p \doteq q \mid p \not\doteq q.$$

Simple constraint formulas will play a role in Section 4.

Let $\mathcal{R}$ denote a set of *relational symbols* (or predicate symbols), where every relation symbol comes with a natural number specifying the number of arguments it takes. Now the constraint language $\mathcal{L}$ will be extended to the relational language $\mathcal{R}(\mathcal{L})$ as follows.

The variables of $\mathcal{R}(\mathcal{L})$ are the variables of $\mathcal{L}$. The so called *relational formulas* of $\mathcal{R}(\mathcal{L})$ are defined inductively. Let r denote a relational symbol in $\mathcal{R}$, x a variable, $\vec{s}$ a tuple of terms which has as many elements as r arguments and ϕ a constraint formula in $\mathcal{L}$. We form relational formulas F, G according to the following syntax rule:

$$F, G \longrightarrow \phi \mid \emptyset \mid r(\vec{s}) \mid F \;\&\; G \mid F \to G \mid \exists x.F.$$

We call $r(\vec{s})$ a *relational atom*.

By extending an interpretation for constraint formulas in a natural way, we get an interpretation for relational formulas. An interpretation $\mathcal{N}$ of $\mathcal{R}(\mathcal{L})$ is obtained from an $\mathcal{L}$-interpretation $\mathcal{I}$ by taking the same domain ($\Delta^{\mathcal{N}} := \Delta^{\mathcal{I}}$) and by choosing for every relation symbol $r \in \mathcal{R}$ with arity n a relation $r^{\mathcal{N}}$ on $(\Delta^{\mathcal{I}})^n$.

Let $\alpha(\vec{s})$ denote $(\alpha(s_1), \ldots, \alpha(s_n))$ for $\vec{s} = (s_1, \ldots, s_n)$. Every $\mathcal{N}$-assignment α satisfies $\emptyset$. Furthermore, we say that α satisfies

$r(\vec{s})$,	if $\alpha(\vec{s}) \in r^{\mathcal{N}}$
$F \;\&\; G$,	if α satisfies F and G
$F \to G$,	if α does not satisfy F or α satisfies G
$\exists x.F$,	if there exists an $\mathcal{N}$-assignment β that satisfies F and $\alpha(y) = \beta(y)$ for all $y \neq x$.

Valid relational formulas and models for relational formulas are defined as in the case of constraint formulas.

Next we introduce a form of rules which allow us to define new relations. A *definite clause* is a relational formula of the form:

$$r(\vec{s}) \leftarrow r_1(\vec{s}_1) \;\&\; r_2(\vec{s}_2) \;\&\; \ldots \;\&\; r_n(\vec{s}_n) \;\&\; \phi$$

where the $r(\vec{s}), r_i(\vec{s}_i)$ are relational atoms and ϕ is a constraint formula.

Let S denote a set of definite clauses. A relational symbol r is called a *basic* symbol if it does not occur on the left hand side of a clause in S and a *defined* symbol otherwise.

Two $\mathcal{R}(\mathcal{L})$-interpretations for S are called *basically equivalent* if their restrictions to constraint symbols and basic symbols are equal. A partial ordering on the set of all $\mathcal{R}(\mathcal{L})$-interpretations for S is defined by:

$$\mathcal{A} \subseteq \mathcal{B} \;:\Longleftrightarrow\; \mathcal{A} \text{ and } \mathcal{B} \text{ are basically equivalent and } \forall r \in \mathcal{R}.\, r^{\mathcal{A}} \subseteq r^{\mathcal{B}}.$$

A *basic interpretation* for $\mathcal{R}(\mathcal{L})$ is an interpretation which only interprets constraint symbols and basic symbols of $\mathcal{R}(\mathcal{L})$.[2]

Proposition 3.8 *Let S denote a set of definite clauses in $\mathcal{R}(\mathcal{L})$ and B a basic interpretation for $\mathcal{R}(\mathcal{L})$. Then there exists a unique minimal extension of B to a model of S.*

Proof. Consider the following construction. Let $\mathcal{N}_0$ be the $\mathcal{R}(\mathcal{L})$-interpretation which extends B such that $r^{\mathcal{N}_0} := \emptyset$ for all defined symbols $r \in \mathcal{R}$. Now we define $\mathcal{R}(\mathcal{L})$-interpretations $\mathcal{N}_{i+1}$ by the following equation:

$$r^{\mathcal{N}_{i+1}} := \{\alpha(\vec{s}) \mid (r(\vec{s}) \leftarrow G) \in S \text{ and } \alpha \text{ is an } \mathcal{N}_i\text{-assignment which satisfies } G\}.$$

Since basic symbols do not occur on the left hand side of the clauses in S, this defines a chain $\mathcal{N}_0,\ \mathcal{N}_1, \ldots$ of basically equivalent $\mathcal{R}(\mathcal{L})$-interpretation. By induction on i one verifies that $\mathcal{N}_i \subseteq \mathcal{N}_{i+1}$. It is easy to see that $\mathcal{N} := \bigcup_{i \geq 0} \mathcal{N}_i$ defines a model of S extending B and that $\mathcal{N}$ is minimal w.r.t. the above defined ordering on $\mathcal{R}(\mathcal{L})$-interpretations. □

Such an extension exists in particular if we have a basic interpretation which is a model for a set of cover-, inclusion- and disjointness axioms.

In section 4 we will define definite clauses as a part of the definitional knowledge. The above proposition gives us the justification to do this.

4 The Representation of the Configuration Knowledge

As already mentioned in section 2, there are different parts of knowledge for specifying a configuration problem in a structured way. With the *definitional knowledge* **D** and the *integrity constraints* **I** one can describe the technical domain of interest. With a *specification* **S** one formulates a special configuration goal and a *configuration* **C** describes a solution for such a goal. In the following we assume that all sets are finite sets.

[2] The definition of a basic interpretation will be extended in section 4.

4.1 The Definitional Knowledge

In this part we describe the overall classes and relations. The definitional knowledge **D** splits into two subparts, the set *Tax* of taxonomical knowledge and the set *Rel* of relational knowledge. The taxonomical knowledge *Tax* consists of a set of cover axioms of the form (cf. section 3.2):

$$S_0 \doteq S_1 \sqcup \cdots \sqcup S_n$$

with the restriction, that

1. for every sort S there is at most one cover axiom $S \doteq D$ in *Tax*
2. *Tax* is cycle free.

Dropping one of these conditions leads to problems with the satisfiability of constraints as will be explained below.

The inclusion- and disjointness axioms are not part of the definitional knowledge. They do not influence the subsumption hierarchy, but represent necessary conditions for elements to be in a sort and therefore belong to the integrity constraints (see below). In the configuration domain all sorts and their relationships are well known. A sort is always the union of its subsorts. This can be modeled with a set of cover axioms in a natural manner.

The relational knowledge *Rel* consists of a set of definite clauses in $\mathcal{R}(\mathcal{L})$ of the form (cf. section 3.3):

$$r(\vec{s}) \leftarrow r_1(\vec{s}_1) \;\&\; r_2(\vec{s}_2) \;\&\; \ldots \;\&\; r_n(\vec{s}_n) \;\&\; \phi,$$

where the variables on the left hand side are a subset of the variables on the right hand side and ϕ is a simple constraint formula.

These clauses introduce new relations between objects of the domain. In particular, if we have an interpretation of the constraint symbols and the basic symbols of $\mathcal{R}(\mathcal{L})$, *Rel* leads to unique minimal denotations for the defined ones (cf. section 3.3).

From now on an interpretation which interprets the features and only the basic sorts and basic relational symbols is called *basic interpretation* for $\mathcal{R}(\mathcal{L})$. From Proposition 3.8 it follows that for every basic interpretation B there exists a unique minimal extension of B to a model of **D**. Here *Tax* uniquely determines the denotation of the nonbasic sorts.

Thus the definitional knowledge plays the important role of a link between abstract and basic notions. The specification of configuration goals and integrity constraints is given in an abstract language containing arbitrary relational and sort symbols (see below). However the description of a solution, i.e. of a concrete technical system is given in terms of basic relational and sort symbols (see below). Given such a basic description, **D** defines unique (minimal) denotations for all relations and sorts.

4.2 The Integrity Constraints

The integrity constraints **I** express necessary conditions which the components of a configuration have to satisfy or rule out certain combinations of components as impossible. We have three kinds of integrity constraints: *sort conditions, forward-rules* and *denials*.

Sort conditions describe necessary conditions for elements of the sorts. They consist of a set of inclusion axioms and disjointness axioms of the form (cf. section 3.2):

$$S \sqsubseteq D, \qquad S_1 \parallel S_2.$$

The sort conditions together with *Tax* have to be admissible in order to get a decidable satisfiability problem for constraints.

By means of sort conditions we can explicitly introduce and/or forbid features. The inclusion axiom

$$S \sqsubseteq f_1.D_1 \sqcap f_2.D_2 \sqcap f_3\uparrow$$

for example introduces the features f_1 and f_2 und forbids the feature f_3 for elements which are in S. Note that the inclusion axiom

$$S \sqsubseteq P(f, g)$$

with P a concrete domain predicate also introduces the features f and g.

Together with the relational part *Tax* of $\mathbf{D}$, subsorts inherit necessary conditions from their supersorts. If we have for example the axioms

$$S \doteq S_1 \sqcup S_2, \qquad S \sqsubseteq f.D_1 \sqcap g\uparrow,$$

then all elements of A_1 and A_2 have a feature f and have no feature g.

Forward rules describe necessary conditions for configurations and have the form:

$$R(\vec{s}) \;\&\; \phi \rightarrow \exists \vec{y}\; Q(\vec{s}, \vec{y}) \;\&\; \psi,$$

where $R(\vec{s})$, $Q(\vec{s}, \vec{y})$ are conjunctions of relational atoms, ϕ is a simple constraint formula, ψ a constraint formula and $\vec{y}$ is a tuple of distinct variables which occur neither in $R(\vec{s})$ nor in ϕ.

Sort conditions and forward rules are called *active* constraints. This plays a role in the calculus for generating a model of $\exists.\mathbf{S}$ and $\mathbf{I}$ (see section 5). If we have an inclusion axiom

$$S \sqsubseteq D$$

and during the generation we put an element a into S, then we try in the following to fulfill all conditions specified in D for a. Or if we have a forward rule and during the generation the left hand side of the rule becomes valid, we try to make the right hand side valid, too.

The denials describe forbidden combinations of components. They have the form:

$$r_1(\vec{s}_1) \;\&\; r_2(\vec{s}_2) \;\&\; \ldots \;\&\; r_n(\vec{s}_n) \;\&\; \phi \rightarrow \bot,$$

where the $r_i(\vec{s}_i)$ are relational atoms and ϕ is a simple constraint formula.

Denials are called *passive* constraints. This notion again comes from the calculus. If we have a denial d and during the model generation process the left hand side of d becomes valid, then we have a clash and must backtrack.

The reason why the constraint formulas in the antecedent of the definite clauses, the forward rules and the denials have to be simple is the following. During the generation of the

configuration by the calculus we have to decide whether such a rule is "firing" or not. We do this more or less by matching syntactically the antecedent with the current partial configuration.[3] We must be sure that we come to the same decision in every configuration derivable from the current one. If we had constraints of the form $s\!:\!p\!\uparrow$ or $s\!:\!\neg S$, we would have to make all negative information explicit. (Absence of positive information doesn't mean presence of negative information and vice versa). In order to avoid this we allow only simple constraints.

We do not forbid constraints of the form $s \neq t$. This would restrict the expressiveness of the description language too much. We handle this case by adding a rule to the calculus which nondeterministically separates objects (by introducing a constraint $s \neq t$) or identifies them (by introducing a constraint $s \doteq t$). Analogous rules would be necessary if we wanted to handle negative constraints of the form $s\!:\!p\!\uparrow$ or $s\!:\!\neg S$. But this nondeterminism would lead to an enormous growup of complexity.

Forward rules and denials can influence the taxonomy. Consider the following forward rule:

$$x\!:\!S_1 \rightarrow x\!:\!S_2.$$

This is semantically equivalent to the terminological axiom $S_1 \sqsubseteq S_2$. Or consider the following denial:

$$x\!:\!S \rightarrow \bot.$$

This is semantically equivalent to the terminological axiom $\neg S \doteq \top$. Even if we forbid denials or forward rules without relational atoms we get interactions between the taxonomy and the integrity constraints. Consider the following example. As forward rule and denial respectively we have:

$$x\!:\!S_1 \rightarrow r(x), \qquad r(x)\ \&\ x\!:\!S_2 \rightarrow \bot.$$

This is semantically equivalent to the terminological axiom $\neg S_1 \sqcup \neg S_2 = \top$.

So we cannot prevent interactions between the taxonomy and the integrity constraints. This leads in general to undecidability of the satisfiability of constraints.

The reason why we insist nonetheless on a decidable taxonomical language is the following. The taxonomical knowledge, i.e. the cover, inclusion and disjointness axioms describe the classes of the domain. If we have a decidable taxonomic language, we can be sure to detect inconsistencies in the description of the domain. This can be very helpful for the knowledge engineer.

4.3 The Specification and the Configuration

With the specification **S** we describe a configuration goal. This is done in an abstract language which allows nonbasic sorts and relations. The specification **S** splits into two subparts, the part SC, which consists of a set of constraints of the form (cf. section 3.2):

$$s\!:\!D,\ spt,\ s \doteq t,\ s \neq t,\ P(s_1, \ldots, s_k),$$

and the part SR, which is a set of relational atoms of the form:

$$r(\vec{s}).$$

[3] See section 5 for more details.

Such a specification is the input of a configuration problem solver. The task of the problem solver is to compute a structure—the configuration **C**—such that

- $\mathbf{C} \models_{\mathbf{D}} \exists.\mathbf{S}$
- $\mathbf{C} \models_{\mathbf{D}} \mathbf{I}$.

S can be seen as a set of relational formulas $F_1, \ldots, F_n$. By $\exists.\mathbf{S}$ we mean the existential closure of **S**, i.e. the formula $\exists x_1 \cdots \exists x_m \; F$, where $F = F_1 \;\&\; F_2 \;\&\; \cdots \;\&\; F_n$ and $x_1, \ldots, x_m$ are all variables appearing in F.

According to section 3.3, an $\mathcal{R}(\mathcal{L})$-interpretation $\mathcal{N}$ is a model of **I** if every integrity constraint is valid in $\mathcal{N}$, i.e. every $\mathcal{N}$-assignments α satisfies every integrity constraint. $\mathcal{N}$ is a model of $\exists.\mathbf{S}$ if $\exists x_1 \cdots \exists x_m \; F$ is valid in $\mathcal{N}$. It is easy to see that this is the case if there exists an $\mathcal{N}$-assignments α which satisfies every relational formula in **S**.

The configuration **C** is represented by a set C of basic relational atoms $r(\vec{s})$ and constraints of the form:

$$s\colon S, \;\; sft,$$

where S is a basic sort.

C describes a basic interpretation **C** in the following sense. For basic sorts A, basic relational symbols r and feature symbols f, **C** is defined as follows:

- $\Delta^{\mathbf{C}} := \{s \mid s \text{ is a variable or constant in } C\}$
- $A^{\mathbf{C}} := \{s \mid s : A \in C\}$
- $r^{\mathbf{C}} := \{(s_1, \ldots, s_n) \mid r(s_1, \ldots, s_n) \in C\}$
- $(s,t) \in f^{\mathbf{C}}$, if $sft \in C$.

According to Proposition 3.8 we get a minimal extension $\tilde{\mathbf{C}}$ of **C** which is a model of **D**. In the next section we will introduce a calculus which computes with input **D**, **I** and **S** a C such that $\tilde{\mathbf{C}}$ is also a model of $\exists.\mathbf{S}$ and **I**.

There we will extract an $\tilde{\mathbf{C}}$-assignment α from C which satisfies obviously every relational formula in **S**. By considering the calculus we will make it also obvious that all integrity constraints are valid in $\tilde{\mathbf{C}}$.

5 A Calculus for Model Construction

Our aim in this section is to give a calculus for computing a structure **C** to given **D**, **I** and **S**. Also we want to show that the calculus is correct, i.e. if the computation stops without an obvious contradiction $\tilde{\mathbf{C}}$ is a model of $\exists.\mathbf{S}$ and **I**. Furthermore, if there exists a finite model, the calculus will find it. But it may happen, that the calculus runs infinitely if there is no finite model or no model at all.

5.1 Preliminaries

Before we give the calculus we have to establish some definitions. The first ones have to do with concrete domain constraint solving.

Let $\mathcal{D}$ denote an admissible concrete domain. A *cd-substitution* σ is a mapping from the set of variables to the set of concrete constants. Let X denote a set of variables. With $\sigma_{|X}$ we denote the restriction of σ to X. Let A denote a set of concrete atoms $P_i(\vec{s}_i)$. We define the *solutions* of A by

$$sol(A) := \{\sigma \mid P_i(\vec{s}_i)\sigma \text{ is true in } \mathcal{D} \text{ for all } P_i(\vec{s}_i) \in A\}.$$

Furthermore, we define

$$sol(A)_{|X} := \{\sigma_{|X} \mid \sigma \in sol(A)\}.$$

For a set of relational formulas G we denote with $cd(G)$ the set of all concrete atoms $P(\vec{s})$ in G. We say G *logically implies* a concrete atom $P(\vec{s})$ (written $G \models P(\vec{s})$), if for every cd-substitution $\sigma \in sol(cd(G))$, $P(\vec{s})\sigma$ is true in $\mathcal{D}$[4]. It is easy to see that $G \models P(\vec{s})$ if and only if $sol(cd(G) \cup \neg P(\vec{s})) = \emptyset$. Notice that since $\mathcal{D}$ is admissible, $\neg P(\vec{s})$ can be represented in $\mathcal{D}$ and therefore logical implication is decidable.

In order to handle solutions in the calculus, the concrete domain has to fulfil some further requirements.

For every set A of concrete domain atoms it must be decidable whether $sol(A)$ is empty, finite or infinite. Additionally there must exist a procedure which enumerates the set $sol(A)$, i.e., returns a first $\sigma \in sol(A)$ to the first call, a second one to the second call and so on. If all σ's are enumerated, the procedure will return `fail`.

The next definitions have to do with forward rules, denials and definite clauses.

A *substitution* σ is a mapping from the set of variables to the set of constants and variables. An *instance* of a relational formula F is a formula we get by applying a substitution σ to F in such a way that every free occurrence of a variable x is replaced by $\sigma(x)$. We write this as $F\sigma$. We say a definite clause, forward rule or denial is *active* w.r.t. a set C of relational formulas, if its antecedent is active w.r.t. C. A formula F & G is *active* w.r.t. C, if F and G are both active w.r.t. C. A relational atom $r(\vec{s})$ is *active* w.r.t. C, if $r(\vec{s}) \in C$. A constraint φ is *active* w.r.t. C if one of the following holds:

- $\varphi = spt$ and $spt \in C$ or there are t_i, $i = 2..n$ with $sf_nt_n, t_nf_{n-1}t_{n-1}, \ldots, t_2f_1t \in C$ and $p = f_1f_2 \cdots f_n$
- $\varphi = s \doteq t$ and $s = t$
- $\varphi = s \not\doteq t$ and either $s \not\doteq t \in C$ or s, t are constants with $s \neq t$
- $\varphi = P(s_1, \ldots, s_k)$ and $C \models P(s_1, \ldots, s_k)$.
- $\varphi = s\colon D$ and $s\colon D \in C$ or $D = D_1 \sqcap \cdots \sqcap D_n$ and $s\colon D_i$, $i = 1..n$ active w.r.t. C
- $\varphi = s\colon \top$

[4]This implication is the "normal" logical implication since our concrete domains come with just one interpretation and the set of all cd-substitutions represents the set of all assignments.

- $\varphi = s{:}\, p \doteq q$ and there is a t with spt and sqt active w.r.t. C
- $\varphi = s{:}\, p \not\doteq q$ and there are s, t with spt, sqt and $s \not\doteq t$ active w.r.t. C
- $\varphi = s{:}\, f.S$ and there is a t with $sft \in C$ and $t{:}\, S \in C$

Notice that the constraints appearing in the antecedents are always simple. With this definition of "active" we want to assure two things. First, that a formula which is valid in the computed structure $\tilde{\mathbf{C}}$ is active at some point of the generation process, and second, that a formula which is active at some point of the generation will be active at every point after.

5.2 The Calculus

We give a set of so-called *completion rules* which act on a tuple $G.C$ with G and C sets of relational formulas. We call $G.C$ a *configuration state*. We always start with the configuration state $\mathbf{S}.\emptyset$. If no more rule is applicable, we call $G.C$ *complete*. With $G.C\sigma$ we denote the configuration state obtained from $G.C$ by replacing each occurence of a variable x by $\sigma(x)$ in G and C.

The *deletion rule* is:

D1: $G \cup \{F\}.C \to G.C$
if F is a relational formula in C

The *sort rules* are:

S1: $G \cup \{s : S\}.C \to G \cup \{s : S_i\}.C \cup \{s : S\}$
if $S \doteq S_1 \sqcup \cdots \sqcup S_n \in \mathbf{D}$

S2: $G \cup \{s : S\}.C \to G.C \cup \{s : S\}$
if S is a basic sort

S3: $G.C \to G \cup \{s : D\}.C$
if $s : S \in C$, $S \sqsubseteq D \in \mathbf{I}$, and $s : D \notin G \cup C$

S4: $G.C \to G.C \cup \{s : \bot\}$
if $S_1 \parallel S_2 \in \mathbf{I}$, $s : S_1 \in C$, $s : S_2 \in C$ and $s : \bot \notin C$

S5: $G.C \to G.C \cup \{s : S'\}$
if $s : S \in C$, $S' \doteq S_1 \sqcup \cdots \sqcup S \sqcup \cdots \sqcup S_n \in \mathbf{D}$, and $s : S' \notin G \cup C$

S6: $G \cup \{s : \neg S\}.C \to G.C \cup \{s : \neg S\}$

S7: $G.C \to G.C \cup \{s : \bot\}$
if $s : S \in C \cup G$, $s : \neg S \in C$ and $s : \bot \notin C$

The *constraint rules* are:

C1: $G \cup \{s : D_1 \sqcap D_2\}.C \to G \cup \{s : D_1, s : D_2\}.C \cup \{s : D_1 \sqcap D_2\}$

C2: $G \cup \{s : D_1 \sqcup D_2\}.C \to G \cup \{s : D\}.C \cup \{s : D_1 \sqcup D_2\}$
if $D = D_1$ or $D = D_2$

C3: $G \cup \{s : p\uparrow\}.C \to G.C \cup \{s : p\uparrow\}$

C4: $G.C \cup \{s : p\uparrow\} \to G.C \cup \{s : \bot\}$
if $sf_nt_n, t_nf_{n-1}t_{n-1}, \ldots, t_2f_1t \in C, p = f_1f_2 \cdots f_n$

C5: $G \cup \{s : \bot\}.C \to G.C \cup \{s : \bot\}$

C6: $G \cup \{s : f.D\}.C \to G \cup \{sfx,\ x : D\}.C \cup \{s : f.D\}$
if x is a new variable

C7: $G \cup \{s : p \doteq q\}.C \to G \cup \{spx,\ sqx\}.C \cup \{s : p \doteq q\}$
if x is a new variable

C8: $G \cup \{s : p \not\doteq q\}.C \to G \cup \{spx,\ sqy,\ x \not\doteq y\}.C \cup \{s : p \not\doteq q\}$
if x and y are new variables

C9: $G \cup \{spft\}.C \rightarrow G \cup \{sfx,\ xpt\}.C \cup \{spft\}$

if x is a new variable

C10: $G \cup \{sft\}.C \rightarrow G.C \cup \{sft\}$

if no $sft' \in C$

C11: $G \cup \{sft\}.C \rightarrow G \cup \{t \doteq t'\}.C$

if $sft' \in C$

C12: $G \cup \{x \doteq s\}.C \rightarrow G.C[x/s]$

C13: $G \cup \{s \doteq x\}.C \rightarrow G.C[x/s]$

C14: $G \cup \{a \doteq b\}.C \rightarrow G.C \cup \{s : \bot\}$

C15: $G \cup \{a \doteq a\}.C \rightarrow G.C$

C16: $G \cup \{s \not\doteq t\}.C \rightarrow G.C \cup \{s \not\doteq t\}$

C17: $G.C \rightarrow G.C \cup \{s : \bot\}$

if $s \not\doteq s \in C$

C18: $G \cup \{s : P(p_1, \ldots, p_k)\}.C \rightarrow G \cup \{P(y_1, \ldots, y_k), sp_1y_1, \ldots, sp_ky_k\}.$
$C \cup \{s : P(p_1, \ldots, p_k)\}$

C19: $G \cup \{P(\vec{s})\}.C \rightarrow (G.C \cup \{P(\vec{s})\})\sigma_{|X}$

if X denotes the set of variables appearing in $\vec{s}$ and $\sigma \in sol(cd(G) \cup P(\vec{s}))$.

C20: $G \cup \{P(\vec{s})\}.C \rightarrow G.C \cup \{s : \bot\}$

if $sol(cd(G) \cup P(\vec{s})) = \emptyset$, s an arbitrary term

The *relational rules* are:

R1: $G \cup \{r(\vec{s})\}.C \rightarrow G \cup \{r_1(\vec{s}_1), \ldots, r_n(\vec{s}_n), \phi\}.C \cup \{r(\vec{s})\}$

if $r(\vec{s}) \leftarrow r_1(\vec{s}_1)\ \&\ \ldots\ \&\ r_n(\vec{s}_n)\ \&\ \phi$ is an instance of a definite clause in $\mathbf{D}$

R2: $G \cup \{r(\vec{s})\}.C \rightarrow G.C \cup \{r(\vec{s})\}$

if r is a basic relational symbol

R3: $G.C \rightarrow G.C \cup \{r(\vec{s})\}$

if $\Gamma = r(\vec{s}) \leftarrow r_1(\vec{s}_1)\ \&\ \ldots\ \&\ r_n(\vec{s}_n)\ \&\ \phi$ is an instance of a clause in $\mathbf{D}$ Γ is active w.r.t. C, and $r(\vec{s}) \notin C$

The *denial rule* is:

D1: $G.C \rightarrow G.C \cup \{s : \bot\}$

if $\Gamma = r_1(\vec{s}_1)\ \&\ r_2(\vec{s}_2)\ \&\ \ldots\ \&\ r_n(\vec{s}_n)\ \&\ \phi \rightarrow \bot$ is an instance of a denial in $\mathbf{I}$ Γ is active w.r.t. C, and $s : \bot \notin C$

The *forward rule* is:

F1: $G.C \to G \cup \{q_1(\vec{s},\vec{y}),\ldots,q_m(\vec{s},\vec{y}),\ \psi_1,\ldots,\psi_n\}.C$

if $\Gamma = R(\vec{s})\ \&\ \phi \to \exists\vec{y}Q(\vec{s},\vec{y})\ \&\ \psi$ is an instance of a forward rule in **I** Γ is active w.r.t. C, $Q(\vec{s},\vec{y}) = q_1(\vec{s},\vec{y})\ \&\ \cdots\ \&\ q_m(\vec{s},\vec{y}), \psi = \psi_1\ \&\ \cdots\ \&\ \psi_n$, there is no $\vec{y}'$ such that $q_1(\vec{s},\vec{y}'),\ldots,q_m(\vec{s},\vec{y}'),\ \psi_1',\ldots,\psi_n'$ $(\psi_i' = \psi_i[\vec{y}/\vec{y}'])$ are in $G \cup C$, and $\vec{y}$ is a tuple of new variables.

The *equation rule* is:

E1: $G.C \to G \cup \{s\rho t\}.C$

if s and t are distinct terms appearing in G, either ρ is $\doteq$ or ρ is $\not\doteq$, and $s\rho t \notin G \cup C$.

We call the rules S1, C2, C19, R1 and E1 *nondeterministic* rules, since they can be applied in different ways to the same $G.C$. All other rules are called *deterministic* rules. We call a configuration state $G.C$ *clash-free* if C does not contain a constraint $s : \bot$.

The behaviour of the calculus can be described as follows. The set G contains the current "goals", whereas C contains the description of the current partial configuration. We start with $G = \mathbf{S}$ and $C = \emptyset$ and eventually end up with a complete $G.C$ with $G = \emptyset$. If $\{s : \bot\}$ is not part of C then C contains a representation of a basic interpretation $\mathbf{C}$ given by constraints of the form : $r(\vec{s})$, $s : S$, sft, with r, S basic, such that $\tilde{\mathbf{C}}$ is a model for $\mathbf{D}$, $\exists.\mathbf{S}$ and $\mathbf{I}$.

But there is more in C. There is a representation of $\tilde{\mathbf{C}}$ in form of constraints: $r(\vec{s})$, $s : S$, with arbitrary r, S, and negative information of the form: $s \not\doteq t$, $s : p\uparrow$, $s : \neg S$. The latter is used to find clashes, the former and the $s \not\doteq t$-constraints are used to determine whether a denial, forward rule or definite clause is active. The former is redundant in the sense that it can be computed from the basic constraints and $\mathbf{D}$. There are also constraints of the form $s : D$ and spt in C. They are necessary in order to avoid applying rules like S3 or F1 infinitely many times. One could think about changing the definition of configuration state to $G.C.E.N.D$ to denote goals, basic information, extension information, negative information and information about descriptions or to $G.C.R$ to denote goals, basic information and rest. But we think—since the membership to a group is uniquely determined—the chosen notation is the best.

Now we want to discuss the different groups of rules.

The deletion rule deletes already reached goals. It prevents us from doing things twice.

The sort rules make explicit the membership of elements to sorts and the sort conditions defined in **I**. Rule S1 is nondeterministic in the sense that it makes a hypothesis about the subclass an element in the superclass belongs to. Here it is important that every class is the union of its subclasses. Rule S2 has an abductive character because it abduces an element from a basic sort.

The constraint rules handle the constraints imposed on elements by the sort conditions in **I** and by the specification **S**. They simplify the constraints until either an obvious

contradiction is reached ($s : \bot \in C$) or one can easily come to an interpretation which satisfies all these constraints. The last three rules handle concrete domain atoms. They assure that in a complete and clash-free configuration state $G.C$ every concrete domain atom that appeared in G is true with the current cd-substitutions. Rule C2 handles the Or-nondeterminism, rule C19 the nondeterminism introduced by multiple solutions of concrete predicates.

The relational rules make explicit the relational knowledge defined in *Rel*. Rule R1 is nondeterministic because there can be different definite clauses in **D** which fulfil the condition. Rule R2 is an abductive rule in the sense that it abduces the validity of the basic relation $r(\vec{s})$. Rule R3 shows the necessity for the variables on the left hand side of a definite clause to be a subset of those on the right hand side. Otherwise this rule could be applied infinitely many times.

The denial and the forward rule assure that all constraints imposed by denials and forward rules in **I** are fulfilled in a complete and clash-free configuration state.

The equation rule is necessary to make all separations of elements and all identifications explicit in order to be able to use the separations in the antecedents of denials, forward rules and definite clauses.

A variant of this calculus is usable as a *configuration checker*. With input **D**, **I**, **S** and **C** it determines whether **C** is a valid configuration w.r.t. **D**, **I** and **S**. We start with the configuration state **S.C**. First we apply the rules S5 and R3. Now we have a complete representation of $\tilde{\mathbf{C}}$ in C. The absence of positive information now means the presence of negative information. Therefore, we can decide the validity of arbitrary constraints, not only of simple ones. We can check now all integrity constraints in **I** for validity. If they are all valid, we use the following rules in order to "deconstruct" constraints and find clashes: S6, S7, C1 to C9, C11 to C18 and C20. Instead of C19 we take a rule which deletes $P(\vec{a})$ from G if $P(\vec{a})$ is true. Since we use a nondeterministic rule (C1), we have to follow several pathes. If no more of the above rules is applicable, we use D1 to delete goals. The input **C** is a valid configuration w.r.t. **D**, **I** and **S** if and only if G is empty after this deletion process and C contains no clash in atleast one of the pathes.

Notice that this calculus is designed for conceptual purposes. It is not intended as a specification for an implementation. Before implementing it, some further considerations are necessary. Two of the main points are *strategy* and *control*. In particular one has to think about the application of the nondeterministic rules. If possible they should be applied only in situations where enough information is available to use them deterministically. Sophisticated heuristics and suitable methods from Constraint Logic Programming like forward checking and look ahead (see [Van89]) are necessary in order to get a system of pratical relevance.

Furthermore, in an implementation it might be unnecessary to make all the information explicit. For example the extension of **C** could remain implicit. Then an inference step for deciding whether formulas are active or not would be necessary.

5.3 Properties of the Calculus

Now we come to some formal propositions about the calculus. First we want to mention that in a complete configuration state there are no more goals left.

Lemma 5.1 *Let $G.C$ be a complete, clash-free configuration state. Then $G = \emptyset$.*

Proof. (Sketch) By contradiction. Assume there is a formula F in G. Analyzing the possible forms of F will lead to a contradiction to the completeness of $G.C$ in every case. □

The application of the rules is *model preserving* in the following sense.

Proposition 5.2 (Invariance) *Let $G.C$, $G'.C'$ be two configuration states. Then:*

1. *If $G'.C'$ is obtained from $G.C$ by application of a deterministic rule, then $\exists.(G \cup C) \cup \mathbf{D} \cup \mathbf{I}$ has a model if and only if $\exists.(G' \cup C') \cup \mathbf{D} \cup \mathbf{I}$ has a model.*
2. *If $G'.C'$ is obtained from $G.C$ by application of a nondeterministic rule, then $\exists.(G \cup C) \cup \mathbf{D} \cup \mathbf{I}$ has a model if $\exists.(G' \cup C') \cup \mathbf{D} \cup \mathbf{I}$ has a model. Furthermore, if a nondeterministic rule applies to $G.C$, then it can be applied in a way that it yields a $G'.C'$ such that $\exists.(G \cup C) \cup \mathbf{D} \cup \mathbf{I}$ has a model if and only if $\exists.(G' \cup C') \cup \mathbf{D} \cup \mathbf{I}$ has a model.*

This proposition can be shown by a tedious analysis of all the rules.

Now we consider the structure $\mathbf{C}$ extractable from a complete and clash-free $G.C$. We show that there are correspondences between semantical and syntactical properties. Notice that a $\tilde{\mathbf{C}}$-assignment α describes also a syntactical renaming where every term s is renamed by $\alpha(s)$. With $[\vec{s}/\alpha(\vec{s})]F$ we denote the formula that we obtain from F if we replace every free occurence of a term s_i in F by $\alpha(s_i)$ with $i = 1..n$ and $\vec{s} = s_1, \ldots, s_n$.

Lemma 5.3 *Let $G.C$ be complete and clash-free. Let $\tilde{\mathbf{C}}$ be the extension of the basic interpretation $\mathbf{C}$ extracted from C. Let α be a $\tilde{\mathbf{C}}$-assignment, ϕ be a simple constraint formula, F an antecedent of a definite clause, a denial or a forward rule. Then the following holds:*

1. *α satisfies $r(\vec{s})$ iff $[\vec{s}/\alpha(\vec{s})]r(\vec{s}) \in C$*
2. *α satisfies ϕ iff $[\vec{t}/\alpha(\vec{t})]\phi$ is active in C with $\vec{t}$ the terms in ϕ*
3. *α satisfies F iff $[\vec{t}/\alpha(\vec{t})]F$ is active in C with $\vec{t}$ the terms in F*
4. *$s \in D^{\tilde{\mathbf{C}}}$ if $s : D \in C$*
5. *$s \in S^{\tilde{\mathbf{C}}}$ iff $s : S \in C$*

This lemma can be shown by an analysis of the involved formulas and descriptions.

The next theorem states the main result of this section, the correctness of the calculus.

Theorem 5.4 (Correctness) *Let $G.C$ be complete and clash-free. Then there exists a model for* $\mathbf{D}$, $\mathbf{I}$ *and* $\exists.\mathbf{S}$.

Proof. (Sketch). The extension $\tilde{\mathbf{C}}$ of the basic interpretation $\mathbf{C}$ extracted from C is a model of $\mathbf{D}$, $\mathbf{I}$ and $\exists.\mathbf{S}$. To see this, consider the following. $\tilde{\mathbf{C}}$ is a model of $\mathbf{D}$ by definition. In order to show that C is a model of $\mathbf{I}$ one has to show that every sort condition, every denial and every forward rule is valid in $\tilde{\mathbf{C}}$. Sort conditions are inclusion and disjointness axioms. To show that an inclusion axiom $S \sqsubseteq D$ is valid, one has to show that $S^{\tilde{\mathbf{C}}} \subseteq D^{\tilde{\mathbf{C}}}$. Assume $s \in S^{\tilde{\mathbf{C}}}$. With Lemma 5.3 we have $s : S \in C$. Then $s : D \in C$ since $G.C$ is complete and with Lemma 5.3 $s \in D^{\tilde{\mathbf{C}}}$. The proof for a disjointness axiom is analogous. For the forward rule one has to show that every $\tilde{\mathbf{C}}$-assignment α, which satisfies the antecedent, satisfies the succedent too. Let α satisfy $R(\vec{s})$ & ϕ. Then, according to Lemma 5.3 $[\vec{s}/\alpha(\vec{s})]R(\vec{s})$ & ϕ is active in C. Since $G.C$ is complete, the rule has "fired" and therefore α satisfies $\exists \vec{y}\, Q(\vec{s}, \vec{y})$ & ψ. The proof for a denial is analogous. As last point one has to show that $\tilde{\mathbf{C}}$ is a model for $\exists.\mathbf{S}$. Consider the $\tilde{\mathbf{C}}$-assignment $\alpha_{\mathbf{C}}$ defined by $\alpha_{\mathbf{C}}(s) := s$. An analysis of the structure of the formulas in C shows, that $\alpha_{\mathbf{C}}$ satisfies C. Since we start with the system $\mathbf{S}.\emptyset$, $\alpha_{\mathbf{C}}$ satisfies S too. □

6 Conclusion

In this paper we have given a logic based formalisation of configuration. We conceive a configuration problem as the task to construct for a given specification, which is understood as a finite set of logical formulas, a model that satisfies the specification.

The language in which configuration problems are specified allows one to express three kinds of knowledge: knowledge about constraints, taxonomies, and rules. The integration of these different representation languages uses the integration schemes of concrete domains in concept languages and of constraint logic programming.

We have described a calculus by which one can compute solutions to configuration problems if they exist and that allows one to recognize that a specification is inconsistent. In particular, the calculus can be used in order to check whether a given configuration satisfies the specification.

The calculus consists of a set of rules that stepwise try to transform a specification into a configuration. It displays a relatively high number of rules. The advantage gained from this fact is a high flexibility in imposing strategies that specify when to apply which rule. Moreover, based on this calculus one can devise procedures that are incomplete in a controlled way by modifying certain rules or by not applying them at all.

The rules come in two variants, deterministic and nondeterministic ones. Of course, in an actual implementation it will be important to avoid nondeterminism whenever possible. This can partly be achieved by preferring a deterministic step to a nondeterministic one whenever there is an alternative. Moreover, it might often be the case that by giving priority to deterministic rules enough information is generated so that a nondeterministic rule can only be applied in one way without immediately leading to an inconsistency.

Still, there are certain shortcomings. Since the calculus essentially tries to construct a model, it will run forever if it is given a specification that has infinite but not finite models.

Moreover, also when given a specification that has a finite model, this will only be found by a fair strategy, that is, a strategy that does not delay an applicable rule infinitely long. Finally, since in an implementation one probably will not want to do exhaustive search, an actual system based on the ideas of constructive problem solving might be incomplete for pragmatic reasons. Whether these are serious drawbacks or not will be an open question as long as our approach is not tested on problems arising in practice.

The next steps to be taken in this research is to express serious examples with our language. If this succeeds one has to elaborate strategies for applying the calculus, which can be done along the ideas discussed above. Of course, in order to do so it is crucial to study the behaviour of the calculus when it is confronted with substantial examples.

References

[BBMR89] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. CLASSIC: A structural data model for objects. In *ACM SIGMOD International Conf. on Management of Data*, pages 58–67, 1989.

[BH91a] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991.

[BH91b] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. Research Report RR-91-10, DFKI Kaiserslautern, 1991.

[BH91c] F. Baader and B. Hollunder. A terminological knowledge representation system with complete inference algorithm. In *Proc. of the Workshop on Processing Declarative Knowledge, PDK-91*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1991.

[BPGL85] R. Brachman, V. P. Gilbert, and H. Levesque. An essential hybrid reasoning system: Knowledge and symbol level accounts in KRYPTON. In *Proc. of the Int. Joint Conf. on Artificial Intelligence*, pages 532–539, Los Angeles, Cal., 1985.

[CG91] R. Cunis and A. Günter. Beiträge zum 5. Workshop "Planen und Konfigurieren". LKI-Report LKI-M-1/91, Labor für Künstliche Intelligenz, Hamburg, 1991.

[CGS90] R. Cunis, A. Günter, and H. Strecker. *Das Plakon Buch.* Informatik-Fachberichte 266. Springer-Verlag, Berlin - Heidelberg - New York, 1990.

[DLNN91a] F. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proc. of the 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning KR-91*, pages 151–162. Morgan Kaufmann, 1991.

[DLNN91b] F. Donini, M. Lenzerini, D. Nardi, and W. Nutt. Tractable concept languages. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence IJCAI-91*, Sydney, 1991.

[Hei88] M. Heinrich. Ein generisches Modell zur Konfigurierung. In J. Hertzberg, editor, *Proceedings of the 3rd Workshop "Planen und Konfigurieren"*, St. Augustin, 1988. GMD-Bericht 388.

[HN90] B. Hollunder and W. Nutt. Subsumption algorithms for concept languages. Research Report RR-90-04, DFKI Kaiserslautern, 1990.

[HS88] M. Höhfeld and G. Smolka. Definite relations over constraint languages. LILOG Report 53, IWBS, Heidelberg, Germany, 1988.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, Munich, Germany, 1987.

[Kle91] R. Klein. Towards a logic-based model representation in configuration problems. In *ÖGAI-91 Workshop on Model Based Reasoning*, Wien, September 1991.

[Kow90] R. Kowalski. Problems and promisses of computational logic. In J. Lloyd, editor, *Symposium on computational logics*, Springer-Verlag, Berlin, 1990.

[Llo90] J. Lloyd, editor. *Symposium on computational logics*, Berlin, November 1990. Springer-Verlag, ESPRIT Basic Research Action Series. Proc. of the ESPRIT Basic Research Action Symposium.

[Mac87] A. K. Mackworth. *Constraint Satisfaction.* Wiley, New York, 1987.

[Mat70] Y. Matijacevič. Enumerable sets are diophantine. *Soviet Math. Doklady*, 11:354–357, 1970. English Translation.

[MB87] R. MacGregor and R. Bates. The Loom knowledge representation language. Technical Report ISI/RS-87-188, University of Southern California, Information Science Institute, Marina del Rey, Cal., 1987.

[Neb89] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems.* PhD thesis, Universität Saarbrücken, 1989. To appear in Springer LNAI.

[QK90] J. Quantz and C. Kindermann. Implementation of the BACK system version 4. Technical Report KIT-Report 78, FB Informatik, Technische Universität Berlin, Berlin, Germany, 1990.

[Sho67] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[Smo88] G. Smolka. A feature logic with subsorts. LILOG Report 33, IWBS, Stuttgart, Germany, 1988.

[SS89] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Springer LNAI 395, 1989.

[SSS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Journal of Artificial Intelligence*, 47, 1991.

[Van89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[Wal87] C. Walther. *A Many-Sorted Calculus based on Resolution and Paramodulation.* Research Notes in Artificial Intelligence. Morgan Kaufmann Publishers, 1987.