Soininen T., **An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks.** Acta Polytechnica Scandinavica, Mathematics and Computing Series. No. 111, Espoo 2000, 209 pp. Published by the Finnish Academies of Technology. ISBN 951-666-560-8. ISSN 1456-9418.

Keywords: Product configuration, knowledge representation, reasoning

Abstract

A configurable product is tailored to meet the individual requirements of a customer. For such a product, the configuration task is roughly defined as designing a product individual using a set of predefined component types while taking into account a set of known restrictions in combining the components.

In this work, a configuration ontology, i.e. a conceptualization, for representing knowledge on the component types and the restrictions in an information system was developed. It synthesizes and extends several previous approaches. In addition, three formal languages with declarative semantics that extend logic programs and constraint satisfaction problems with configuration specific constructs were developed. A simplified version of the ontology was formalized using one the languages. The languages and the formalization are novel in that their semantics captures the notion of groundedness, i.e. that a configuration must only contain things that are allowed by a configuration model of the product. This property and the configuration specific constructs made uniform formalization of the ontology straightforward.

The computational complexity of the configuration task and relative expressiveness of the languages and the simplified ontology were analyzed. For most of the languages and the formalization, the task is **NP**-complete. The groundedness property was shown to increase the expressive power. Preliminary empirical evidence for the practical relevance of the approach and feasibility of implementing it was found by modeling simple case products and testing a prototype implementation of the languages on small problems.

The main conclusion of this work is that, although there is no general formal model of product configuration yet, the different approaches can be unified under a formal framework. However, the ontology, languages and implementation should all be further empirically tested and validated. Several topics for their further development are also pointed out.

Supervisor

Professor Reijo Sulonen
Department of Computer Science and Engineering
Helsinki University of Technology
Espoo, Finland

Reviewers

Professor Gerhard Friedrich
Institut für Wirtschaftsinformatik und Anwendungssysteme
Universität Klagenfurt
Klagenfurt, Austria

Professor Jukka Paakki
Department of Computer Science
University of Helsinki
Helsinki, Finland

Opponent

Professor Markus Stumptner
Institut für Informationssysteme
Technische Universität Wien
Wien, Austria

Contents

Acknowledgements

This thesis was prepared in the Product Data Management Group (PDMG) of the HUT. I gratefully acknowledge the financial basis for it, the graduate assistantship of Helsinki Graduate School in Computer Science and Engineering (HeCSE). Research project funding from the Technology Development Centre of Finland and the Academy of Finland is also gratefully acknowledged.

I wish to thank all the people of PDMG for the stimulating and fun working environment. The supervisor of this thesis, Professor *Shosta Sulonen*, has done a great job in creating its material infrastructure and spirit. I thank you for the advice, constructive criticism, and especially support during the difficult moments along the way. I am also grateful for instilling in me the respect for real-world practical problems as a guide for research.

Lic. Tech. *Juha Tiihonen* has been a valuable colleague in analyzing the practical issues related to product configuration. Thank you for advising me on research-related matters and the many practical work-related issues. Dr. Sc. *Tomi Männistö* has been a source of excellent constructive criticism and aid in exploring the concepts in this work. I am grateful to you for having the time for the discussions that sometimes seemed to meander along without direction but at other times provided surprising insights. Dr. Sc. *Hannu Peltonen* has also provided invaluable comments and advice on how to best put things down in writing, for which I am grateful. I also thank Research Director *Asko Martio* for providing insight on the industrial practices and on many other practical issues.

Beyond PDMG, I am very grateful to Professor *Ilkka Niemelä* whose excellent teaching on formal methods have inspired me to study these issues. Thank you for the excellent co-operation and practical advice in exploring them. I am also grateful to Ph.D. *Esther Gelle* for her collaboration and the many valuable and enjoyable discussions on thesis and research work we have had. Professor *Martti Mäntylä*, director of HeCSE, has also supported the progress of my thesis and many other aspects of my work, for which I am grateful. I extend thanks to the pre-examiners of my thesis, Professors *Gerhard Friedrich* and *Jukka Paakki*, for their valuable comments and advice. I am also grateful to Ms *Kathleen Tipton* for reviewing the English language of the thesis.

A special thank you goes to *Mia*. Without your patience, support and love this thesis would never have been completed. Thank you for *Juho* as well.

Espoo, November 21, 2000

Timo Soininen

List of Publications

This thesis is based on the following publications, which are referred to in the text by their Roman numerals.

I Niemelä I., Simons P. and Soininen T[1]. Stable Model Semantics for Weight Constraint Rules. In Gelfond, M., Leone, N., and Pfeifer, G. (ed.) *Logic Programming and Non-Monotonic Reasoning 5th International Conference, LPNMR '99, Proceedings*, Springer-Verlag. Pages 317-31. 1999.

II Soininen T., Gelle E. and Niemelä I. A Fixpoint Definition of Dynamic Constraint Satisfaction. In Jaffar, J. (ed.) *Principles and Practice of Constraint Programming- CP'99 5th International Conference, CP'99, Proceedings*, Springer-Verlag. Pages 419-33. 1999.

III Soininen T. and Niemelä I. Developing a Declarative Rule Language for Applications in Product Configuration. In Gupta, G. (ed.) *Practical Aspects of Declarative Languages: First International Workshop, PADL'99, Proceedings*, Springer-Verlag. Pages 305-19. 1999.

IV Soininen T., Niemelä I., Tiihonen J. and Sulonen R. Unified Configuration Knowledge Representation Using Weight Constraint Rules. Technical Report B149. Laboratory of Information Processing Science, Helsinki University of Technology. 2000. Presented at the *ECAI00 Workshop on Configuration*, 21-22 August, 2000.

V Soininen T., Tiihonen J., Männistö T. and Sulonen R. Towards a General Ontology of Configuration. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing,* 12(4):357-72. 1998.

VI Tiihonen J., Lehtonen T., Soininen T., Pulkkinen A., Sulonen, R. and Riitahuhta A. Modeling Configurable Product Families. In Tichem, M. et al (eds.) *Proceedings of the 4th WDK Workshop on Product Structuring*, October 22-23, 1998, Delft University of Technology. Pages 29-50. 1998.

[1] Authors are given in alphabetical order for this publication.

1 Introduction

In this section, the background of the thesis is first covered in Section 1.1 by taking a brief look at the phenomenon of product configuration and the previous related research. The research problems and goals of the research are then enumerated in Section 1.2. The method of research is described in Section 1.3 and the scope of the thesis outlined in Section 1.4. Finally, the structure of the rest of the thesis is given in Section 1.5.

1.1 Background

The design and production of goods that satisfy the specific needs of individual customers are of central interest to many companies. Some major trends in the business environment of these companies include diminishing lifetimes of products[2] and pressure for shorter lead-times in the sales-delivery process. Moreover, there is increasing pressure to adapt product individuals according to customer requirements. This leads to increasing complexity and an increasing number of variants of products.

One way to cope with the changes in the business environment is to develop and deliver *configurable products*. Such products can be characterized by the following properties (Tiihonen et al. 1998; Tiihonen et al. 1996):

- Each delivered product individual is adapted to meet the requirements of a customer.
- The product has been pre-designed to meet a given range of different customer requirements. It is not meant to be adapted to meet requirements outside this range.
- Each product individual is specified as a combination of pre-defined component types according to known rules. New component types are not designed in the sales-delivery process to adapt the product.
- The product has a pre-designed general architecture.
- The adaptation in the sales-delivery process requires only routine design (Tong and Sriram 1992) and can be done in a systematic manner.

[2] In the following, the term “product” encompasses both physical and logical products, such as software, and services.

A configurable product typically has a large number of variants. As the set of components and the rules of adaptation are known, a configurable product allows systematic adaptation, *configuration*, through which the customer-specific variants are generated. The goal is to do this while keeping the configuring of the product inexpensive, cost-effective, and the lead-time of the sales-delivery process short. In other words, a configurable product aims at combining some of the benefits of mass-produced, fixed products, such as relatively low price and short delivery time, and one-of-a-kind products, such as adaptability to customer requirements. This type of operation has sometimes been classified as mass-customization (Hales 1992; Carson 1997).

A configurable product may be very complex to configure due to a large number of components and large number of or highly complex interdependencies of the rules of adaptation. It is also often the case that the product is incompletely or unclearly documented. These aspects lead easily to errors in configuring the product. The errors can be to some extent eliminated by applying a *product configurator*[3], or *configurator* for short, an information system that configures a product or supports a person in doing it.

The *configuration task* carried out by a *configuration engine* in a configurator can be roughly defined as the problem of designing a product individual using a set of pre-defined component types while taking into account a set of well-defined rules on how the components can be combined (**Figure 1.1**). The inputs of the problem are a *configuration model*, which describes the components that can be included in the configuration and the rules on how they can be combined to form a working product, and *requirements* that specify some properties that the product individual should have. The output is a *configuration*, an accurate enough description of a product individual. The configuration must be *suitable*, i.e. satisfy the requirements, and *correct*, i.e., be composed of the components in the configuration model and not break any of the rules in it.

Product configurators have been used as an aid in the sales-delivery process at least from the beginning of the 1980s (McDermott 1982). In the last five years, the number of vendors of product configurators has bloomed (Richardson 1997). The commercial importance of the field is also witnessed by the growing number of companies that have taken a configurator into use (Richardson 1997; Faltings and Freuder 1998). In a survey of ten manufacturing companies it was found that there is a clear need for product configurators

[3] The word "configurer" is sometimes used with the same meaning. There seems to be no standard terminology. The word "configurator" is used in this thesis for an information system. The word "configurer" is reserved for a person that does the configuration task, possibly supported by a configurator.

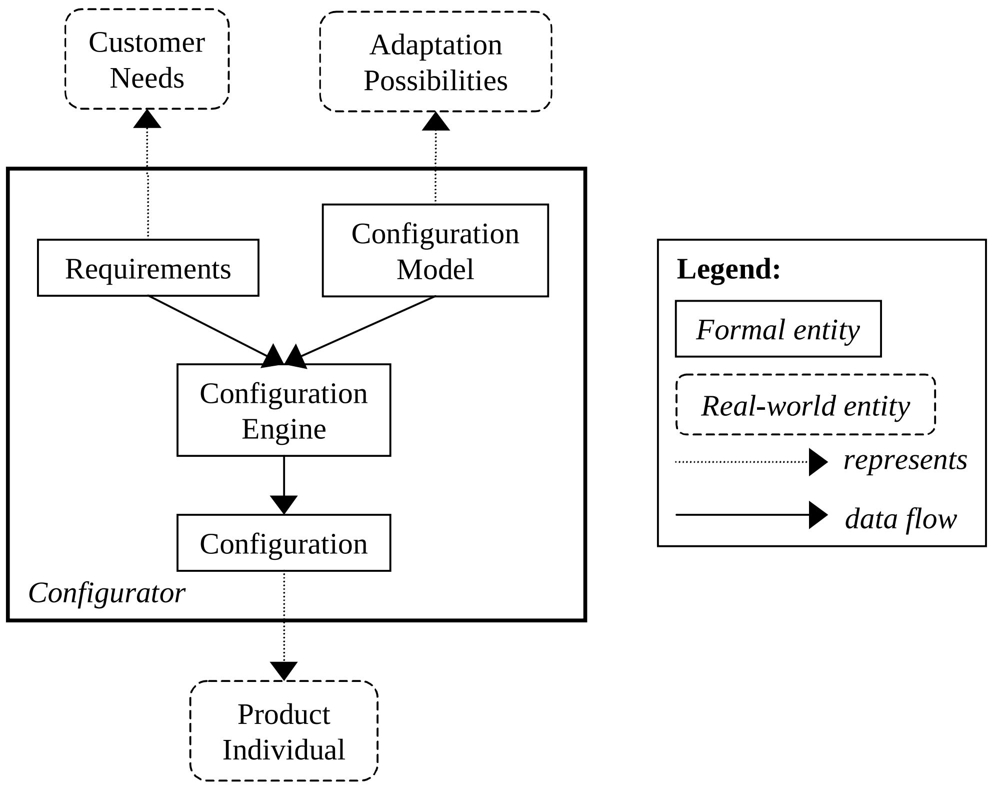


Figure 1.1 Configuration task carried out by a configurator

(Tiihonen et al. 1998; Tiihonen et al. 1996). However, there is no generally accepted commercial or even theoretical solution that would cover the different products and needs of companies.

Theoretical models of configuration tasks and product configurators have been developed within the field of artificial intelligence (AI) for at least two decades, motivated by the practical importance. Representing the *configuration knowledge*, i.e., configuration models, requirements and configurations, is a natural target of the knowledge representation subfield of AI. The configuration task fits into the AI-typical formulation of a task as a search for a solution through a search space of possible configurations. This thesis is a further study in representing configuration knowledge and the reasoning for configuration tasks in computers.

Numerous diverging theoretical models of product configuration tasks and reports on implemented configurators have been presented in the proceedings of configuration workshops (Faltings and Freuder 1996; Faltings et al.

1999; Baader et al. 1996), special issues on configuration (Faltings and Freuder 1998; Darr et al. 1998) and surveys (Sabin and Weigel 1998; Schreiber and Birmingham 1996; Schreiber and Wielinga 1997). The models and reports on implemented configurators have usually addressed (at least some of) the problems of

- developing a conceptualization (explicit or implicit) and formal languages for representing the configuration knowledge,
- developing algorithmic solutions, i.e. *problem solving methods*, for carrying out or supporting configuration tasks by configuration engines,
- implementing the languages and algorithmic solutions as efficiently as possible, and
- gathering empirical evidence of the suitability of the models and methods to real-world configuration problems.

Most of the research has concentrated on applying problem solving methods, such as constraint satisfaction and its extensions (Mittal and Falkenhainer 1990; Stumptner et al. 1998; Gelle 1998; Weigel and Faltings 1996; Sabin and Weigel 1998), and propose-and-revise type approaches (Balkany et al. 1993; Schreiber and Birmingham 1996). Other approaches have defined specific configuration domain-oriented conceptualizations. These include the three main conceptualizations of configuration knowledge as resource balancing (Jüngst and Heinrich 1998), product structure (e.g. (Cunis et al. 1989)) and connections within a product (Mittal and Frayman 1989).

Rule- and knowledge-based systems (KBS) employing the problem solving methods and relatively well-understood general formalisms, such as constraint satisfaction, its extensions, description logics and logic programs have been applicable to real-world product configuration tasks (Faltings and Freuder 1996). This is due to the well-defined, complete configuration knowledge and well-defined nature of the configuration task, which is in contrast to several other application domains where AI has yet to demonstrate significant success.

Despite the research no widely accepted theoretical model of configuration knowledge and tasks that would cover all the relevant aspects in a satisfactory manner has emerged. The approaches have radically different viewpoints on configuration knowledge. The conceptualizations underlying the knowledge representation and the reasoning required for configuration on the basis of the knowledge in these approaches have little in common.

A general conceptualization of the configuration knowledge would be useful to re-use and share configuration models between researchers and product configurators based on the different methods. Such a general model of configuration knowledge is an equally important research issue as the problem solving methods. These two issues are connected, as the conceptual model affects the computational methods that can be used to carry out the configuration task and vice versa, and they should be given equal attention.

Most of the models presented also lack a sound formal basis that would allow a rigorous analysis of the configuration knowledge and tasks, comparison of the models and their implementations, and developing efficient implementations, although this has been to some extent remedied in the recent research. The theoretical models presented so far have widely diverging viewpoints on configuration. Notable formal models include those developed in (Najman and Stein 1992), (Klein et al. 1994), (Gruber et al. 1996), (Felfernig et al. 1999), some of the approaches in (Baader et al. 1996) and the constraint-based approaches mentioned above. Despite the research, few formal models of configuration unifying the different formal and conceptual approaches exist. The formal models presented accommodate only some aspects of the product configuration problem, while the more extensive models are less formal. The diversity of approaches also holds for the implemented systems.

The differences in the models, lack of formal, unifying models and the practical significance of the field make further research on the theoretical basis of configuration knowledge, configuration tasks and product configurators important.

1.2 Research Problem and Goals

The problem that this work addresses is twofold:

- Is there a unified formal conceptualization for representing the knowledge in real-world configuration models, requirements and configurations?
- How hard are the configuration tasks computationally?

The problems are related to developing a conceptual and computational model of real-world configuration tasks. The primary goals of this work were to:

- Develop an explicit, detailed model of the *configuration concepts* that can be used to represent configuration models, configurations and requirements. The model should enable accurate communication of the knowledge and its computer-based manipulation. It should synthesize the previous research and extend it according to experience with real products. (Addressed in V, VI.)

- Develop formal models of the configuration tasks, i.e. of the computation and reasoning required in them. The models should support representing configuration models based on the configuration concepts. In addition, they should be analyzed from the computational complexity point of view to characterize how difficult the configuration tasks are (Addressed in I-IV).
- Provide empirical evidence of the feasibility of the approach taken by modeling real products and developing the foundation of a prototype that carries out configuration tasks based on the formal models. (Addressed in I-III and VI.)

1.3 Method

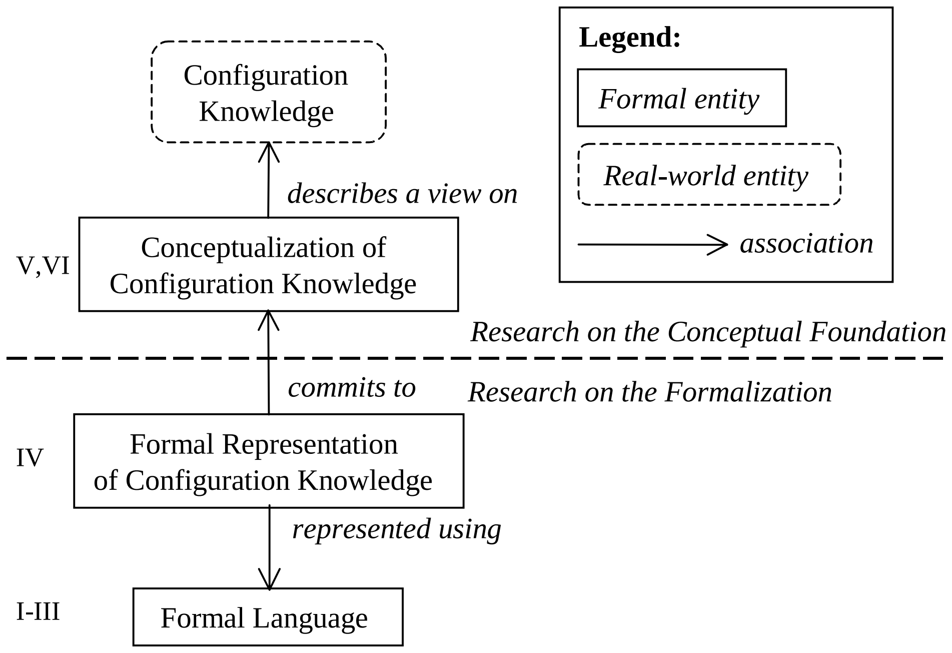
The approach to configuration knowledge representation taken in this work was two-layered (**Figure 1.2**): at the top layer, the conceptualization of configuration knowledge was analyzed and developed, while concurrently at the bottom layer formal declarative languages for formalizing the top layer were developed. The development of the bottom layer was heavily influenced by the results of the top layer. Furthermore, the top layer was partially formalized using a formal language developed at the bottom layer.

The approach taken here is similar to other work in configuration, and in general in the knowledge representation field of AI, where it is sometimes referred to as "logic through the backdoor". The idea is to provide knowledge representation languages that are intuitively understandable to domain experts, such as product designers creating the configuration models, while "hiding" underneath a rigorous formal semantics upon which inference, for example in a configuration engine, can be based.

The two-level approach was adopted for two reasons:

- It was not originally clear what the concepts were nor what their formal semantics would be. Therefore, general formalisms where different concepts and their semantic variations could be tried were desirable.
- By using an implementation of a bottom layer language, fast prototype implementations of the configuration task based on different conceptualizations were achieved.

However, the emphasis of this work was not in developing a particular knowledge representation language for configuration tasks. Rather, the conceptualization underlying the knowledge was analyzed and partially formalized. Several different types of languages, rule-like, object-oriented, etc., could be based on



Legend:
Formal entity
Real-world entity
association
Configuration Knowledge
describes a view on
V,VI
Conceptualization of Configuration Knowledge
Research on the Conceptual Foundation
commits to
Research on the Formalization
IV
Formal Representation of Configuration Knowledge
represented using
I-III
Formal Language

Figure 1.2 The research approach. The numerals refer to the publications in which each area is dealt with.

the same conceptualization. This choice was made to increase the generality of the approach.

The development of the formal models of configuration tasks (I-IV) was carried out within the context of research on knowledge representation and reasoning (KR), which is a sub-field of AI concentrating on representing, maintaining and manipulating knowledge on an application domain (Lakemeyer and Nebel 1994). This is done by developing representation languages for different application purposes and representing knowledge about the application domains using the languages. In addition, reasoning mechanisms on the knowledge represented are specified and the properties of the reasoning analyzed. Finally, systems that implement the languages and the related reasoning are constructed.

The main assumption behind this branch of research is that the knowledge is *explicitly* and *declaratively* represented in a knowledge base consisting of a set of formal entities and whose meaning can be specified without referring to the procedural application of the knowledge. The research often consists of defining appropriate languages, inference methods for inferring implicit knowledge not explicitly represented in the knowledge base, and analyzing the computational properties of the representations and the inference methods. In many cases, a balance must be sought between the expressiveness of the language and its computational properties to guarantee efficiency.

The formal models of configuration tasks were explored by developing a set of progressively more expressive formal languages that can be used to represent typical configuration knowledge elements and by giving each language a formal declarative semantics. The languages considerably extend the previous approaches to configuration knowledge representation, although they are based on some earlier approaches. The languages were defined with the goal that at least some of the concepts in the configuration ontology could be straightforwardly represented in them. In addition, the aim was that it should be possible to represent the rest of the concepts by extending the language.

Techniques developed in non-monotonic reasoning and logic programming research (see, e.g. (Ginsberg 1987; Lloyd 1987; Przymusinska and Przymusinski 1990; Dix 1995)) on providing a declarative semantics for logic program rules of form

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

turned out to be relevant for capturing the *justification* or *groundedness* of the elements in a configuration. This rules are intuitively read as "if $b_1, \ldots, b_m$ are true and $c_1, \ldots, c_n$ are not, then a_1 or … or a_l is true". The research has tried to give an intuitive, common sense formalization of the meaning of such rules or their special cases. This is done by developing and analyzing different formal semantics for what it means that a rule provides a justification for concluding that a_1 or … or a_l is true. Groundedness in a configuration context means that a configuration should not include anything that is not necessary for the product to work, i.e., anything that is not justified by the requirements and the configuration model. This aspect has been identified as an important aspect of configuration knowledge and tasks, and incorporated in limited forms in several approaches (Mittal and Falkenhainer 1990; Stumptner et al. 1998). In contrast, in the formal models presented in this work the groundedness aspect is treated uniformly for all constructs of the languages by equipping each with a formal semantics that encodes it.

The semantics of the languages are thus closely related to the declarative semantics of logic programs. This relation was exploited in developing a prototype implementation of the languages. It is based on extending an existing declarative logic programming system *Smodels* (Simons 2000) and developing translations from the languages in this work to the one supported by it.

The analysis of the conceptualization of configuration knowledge (V) was carried out within the context of research on ontology engineering, a subfield of AI with links to philosophy and linguistics (Gruber 1995; Guarino 1995; Guarino 1997; Guarino and Giaretta 1995). This branch of research aims at developing reusable libraries describing the essential features of a domain of interest. Such libraries are called *ontologies*. They explicitly and formally specify an intensional conceptualization of a domain, i.e. the concepts, their properties and relations to each other that are relevant to reasoning on the domain, independent of a particular problem-solving method used.

The development of the configuration ontology was based on the synthesis of a set of previously presented theoretical models, which had little in common except the central notion of a component. These models were chosen because they all have gained popularity in the research community or have been used in commercial applications. Experience with configurable products gained in projects dealing with the management of configurable products and support for the configuration carried out in co-operation with several industrial companies (Tiihonen 1994; Tiihonen and Soininen 1997; Tiihonen et al. 1998; Tiihonen et al. 1996) indicated that a synthesis of these models is needed to compactly and adequately represent the knowledge on products.

The configuration ontology was defined based on the Frame Ontology of the Ontolingua approach to ontology development (Gruber 1993; Gruber 1992). The basic concepts in Frame Ontology are *classes*, *instances*, *relations* and *functions*. These can be defined to have properties, such as that instances of a particular class are always in some relation or that a particular relation is transitive. Classes collect the definitions that apply to all their instances. A class C_1 is said to be a *subclass* of class C_2 if and only if every potential instance of C_1 is also an instance of C_2. The subclass-relation induces a *taxonomy* in which the definitions are *monotonically inherited*. A *superclass*-relation between classes is defined as the inverse of the subclass-relation in the obvious way. An instance I is said to be a *direct* instance of a class C_1 if there is no class C_2 such that C_2 is a subclass of C_1 and I is an instance of C_2. A *direct* subclass is defined similarly.

A subset of the configuration ontology was also formalized using the most expressive language to illustrate the formalization part of the approach (IV).

The complexities of the main configuration related computational tasks for the languages and the simplified configuration ontology were analyzed using the techniques developed in the field of computational complexity (see, e.g., (Papadimitriou 1994)). Computational complexity provides a theoretical framework for mathematically analyzing how hard a computational problem is to solve by a computer. The major motivation for such analysis is to classify computational problems to those that can be solved efficiently using a computer, those that cannot be solved efficiently and those that cannot be solved at all. A second benefit is to provide a deeper understanding of the problems, information on how they relate to each other, and hints on how to develop efficient algorithms for them or on how the problem should be modified to get an efficient algorithm (Lakemeyer and Nebel 1994). Several simplifying assumptions make the theory independent of the actual computer used to carry out the computation and make the analysis feasible while still providing meaningful information on the nature of the computational problems.

The practical feasibility of the ontology was studied by partially modeling three case products (IV-VI). In addition, in order to estimate the feasibility of the formal models, several simple imaginary configuration problems were represented in the languages, and the efficiency of their implementations tested. (I-III).

1.4 Scope

When developing the conceptualization, only the major approaches that have left a lasting impression on research or were shown practically relevant and provided a clearly defined conceptualization were reviewed, synthesized and compared with the results of this work. In addition, when developing the formal models, only those previous approaches providing a clear formal model were reviewed, used as a basis for developing the formalisms and discussed in relation to this work. Thus, reports describing single use systems not based on a general conceptual model or applications of such systems in a company were not considered relevant to this work.

In developing the configuration ontology, a product configuration point of view was taken instead of considering configuration design in the product development process. The ontology does not cover the geometry, pricing, delivery times and optimality of configurations, which are important for some products. Construction and control knowledge on how to accomplish the configuration task, i.e. the actions and their ordering for configuring a product (Günter et al. 1990), were also excluded. This restriction was made to simplify the problem, especially since the main emphasis in this work is to provide a declarative, non-procedural, representation of configuration knowledge. It is assumed that the missing aspects can be defined on top of the ontology developed in this work.

The formal models of the configuration task and their implementations were also based on a simplification of the real-world configuration tasks and the configuration concepts. It was assumed that the more complex tasks and configuration concepts could be defined on top of this model. The reason for this restriction was again the complexity of the whole problem. Only proof of concept for the approach was sought. The implementations of the formal models were tested on a few small examples only instead of modeling a large set of real world products. This was done to get preliminary results on the feasibility of this approach before proceeding with any more large-scale experiments.

1.5 Outline of the Thesis

The remainder of this thesis is organized as follows: in Section 2 the main results of this thesis are given. In Section 3 the results are discussed and compared with several previous approaches to configuration knowledge representation and reasoning. Further empirical and theoretical work is outlined in Section 4. In Section 5 conclusions on the work are given. The publications annexed to the thesis follow as appendices.

2 A Configuration Ontology and its Formalization

In this section, the main results of the thesis are described. First, the conceptual modeling of configuration knowledge is addressed. In Section 2.1 a generalized informal configuration ontology synthesizing and extending previous work (V, VI) is described in brief. Then, in Section 2.2 a simplification of the generalized ontology (IV) and a simple visual notation to represent configuration knowledge based on it are described in more detail. A simple configuration model of an example product (IV) is also given.

Then, attention is turned to tools for formal representation of configuration knowledge. Three formalisms designed to represent configuration knowledge and implementing the configuration task based on them are presented. The intuitions behind the syntax and semantics of the formalisms are presented and simple examples of modeling products using the formalisms are given. The first two formalisms (Sections 2.3 and 2.4), Configuration Rule Language and Extended Dynamic Constraint Satisfaction Problem, extend normal logic programs (III) and dynamic constraint satisfaction problems (II), respectively, with constructs that are useful in representing configuration knowledge. The constructs allow direct representation of configuration-typical choices between elements in a configuration model, optional choices, and dependencies between the elements in a configuration model. Normal logic programs and dynamic constraint satisfaction problems do not adequately support representing this type of knowledge.

The third formalism, Weight Constraint Rule Language, (I) in Section 2.5 generalizes the first two formalisms. First weighted choices for representing resource interactions are introduced, resulting in a Propositional Weight Constraint Rule Language. Then, this language is further generalized to First-order Propositional Weight Constraint Rule Language by introducing predicate and object symbols which represent the relations between the objects in a configuration. Encoding these types of knowledge using the first three languages would be very cumbersome. A special case of the fourth language, Domain Restricted Weight Constraint Rule Language with restricted use of variables and quantification, is also defined. This language is specifically designed for representing configuration knowledge based on the simplified configuration ontology in a uniform and simple manner, while considering the computational efficiency of the configuration task. A brief explanation of the technique for defining the formal semantics with the groundedness property for the three formalisms is given in Section 2.6. The technical details on the formalisms can be found in (I-III).

Next, the conceptual modeling and formal representation of configuration knowledge are brought together. The configuration model of the case product in Section 2.2 based on the simplified ontology is formalized using the third, most expressive of the developed formalisms (IV) (Section 2.7), providing a synthesis of the representation formalisms and the configuration ontology.

After this, results on the computational complexity of the relevant reasoning tasks for and the expressivity of the formalisms are given in Section 2.8 (I-III). The computational complexities of the configuration related tasks based on the simplified ontology are also analyzed (IV).

Finally, empirical evidence supporting the approach taken in this work is presented in Section 2.9. Simplified models of a further two case products based on the ontology are described to show the applicability of the ontology (IV-VI). A prototype implementation of the formalisms, and hence the ontology, is also briefly described and test results for simple examples given (I-III).

2.1 A Generalized Configuration Ontology

The generalized configuration ontology presented in (V) is a synthesis of the main conceptual approaches to configuration. These approaches are based on modeling the structure of (e.g. (Cunis et al. 1989)), connections within (Mittal and Frayman 1989), and resource interactions (Jüngst and Heinrich 1998) in a product. It consists of a set of concepts for representing the knowledge on a configuration and the restrictions on possible configurations.

There are three main categories of knowledge in the ontology: *configuration solution knowledge*, *configuration model knowledge* and *requirements knowledge*. Configuration solution knowledge specifies a (possibly partial) configuration. Configuration model knowledge specifies the entities that may appear in a configuration and the rules on how the entities can be combined. Requirements knowledge specifies the requirements on a configuration to be constructed. This knowledge can be specified with similar concepts as configuration model knowledge. Therefore, in the following only the representation of configuration model knowledge and configuration solution knowledge is discussed.

The ontology consists of a set of concepts that is introduced to capture configuration model and configuration solution knowledge. The concepts include *components, attributes, resources, ports, contexts, functions, constraints and relations between these*. Components and their compositional structure represent the building blocks out of which a product is constructed and how they are composed of other building blocks. Resources are produced and used by components. Ports represent interfaces at which components can be connected to each other. Contexts represent subsets of a configuration in which resource production and use must be balanced. Different combinations of components produce different functions, which are descriptions of the effects that the product produces in its environment or provides to the customer or user. Components, resources, ports and functions may define attributes that describe their inherent properties. Constraints define conditions that a correct configuration must satisfy.

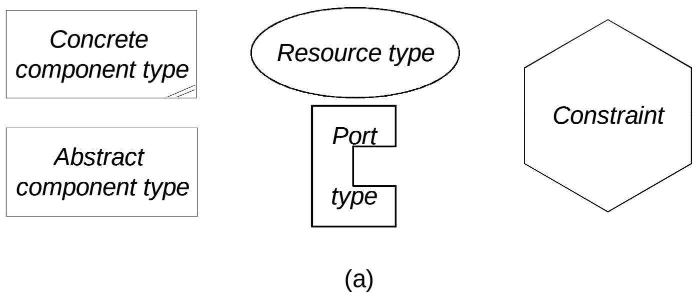
The ontology extends earlier approaches with new concepts arising from practical experience on configurable products. The main extensions are in the detailed conceptualization of knowledge on product structures and in extending the resource concept with contexts for limiting the availability and use of resources. The concepts are treated uniformly in an object-oriented manner with respect to type-instance distinction and classification, which had not been the case in previous work. In addition, constraint sets representing different views on the product are introduced.

2.2 A Simplified Configuration Ontology

In this section, a simplified version of the generalized configuration ontology (IV) is described in more detail. A simple visual notation for the concepts and examples on using the concepts are also given.

2.2.1 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 of types: *component types*, *resource types* and *port types*. In addition, a configuration model includes a set of *constraints*. The types are organized by *IsA-relation* in a *taxonomy* or *class hierarchy* where a subtype *inherits* the properties of its supertypes in the usual manner. For simplicity, only individuals of *concrete*, i.e. leaf, types of the taxonomy that unambiguously describe the product, are allowed in a configuration. In **Figure 2.1**(a) a simple visual notation (VI) for representing the different types and in **Figure 2.1**(b) the notation for taxonomy of a configuration model is given.



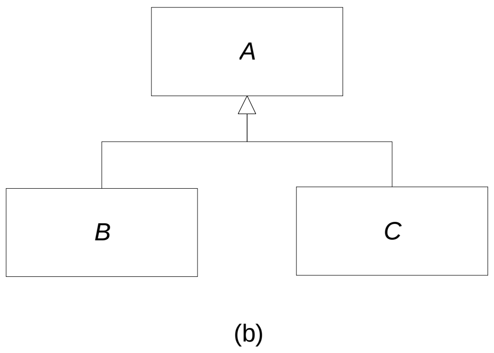


Figure 2.1 (a) Notation for the types and constraints in a configuration model. (b) Notation for the classification hierarchy: Component type *B IsA* component type *A*, i.e. is a subtype of *A*. Similarly for component type *C*.

Example 2.1 A simplified configuration model of a PC in **Figure 2.2** is used to demonstrate the simplified configuration ontology and its formalization. The taxonomy of the PC configuration model is as follows: A PC is represented by component type *pc*, which is a subtype of *cmp*, the generic component type. The 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 *hda* and *hdb*. IDE CD ROM devices *cda* and *cdb* are subtypes of type *cd*, which in turn is a subtype of *ide*. Software packages are represented by type *sw*. Software package types *swa* and *swb* are subtypes of *sw*. Port and resource types in the configuration model are introduced in the following sections.

■

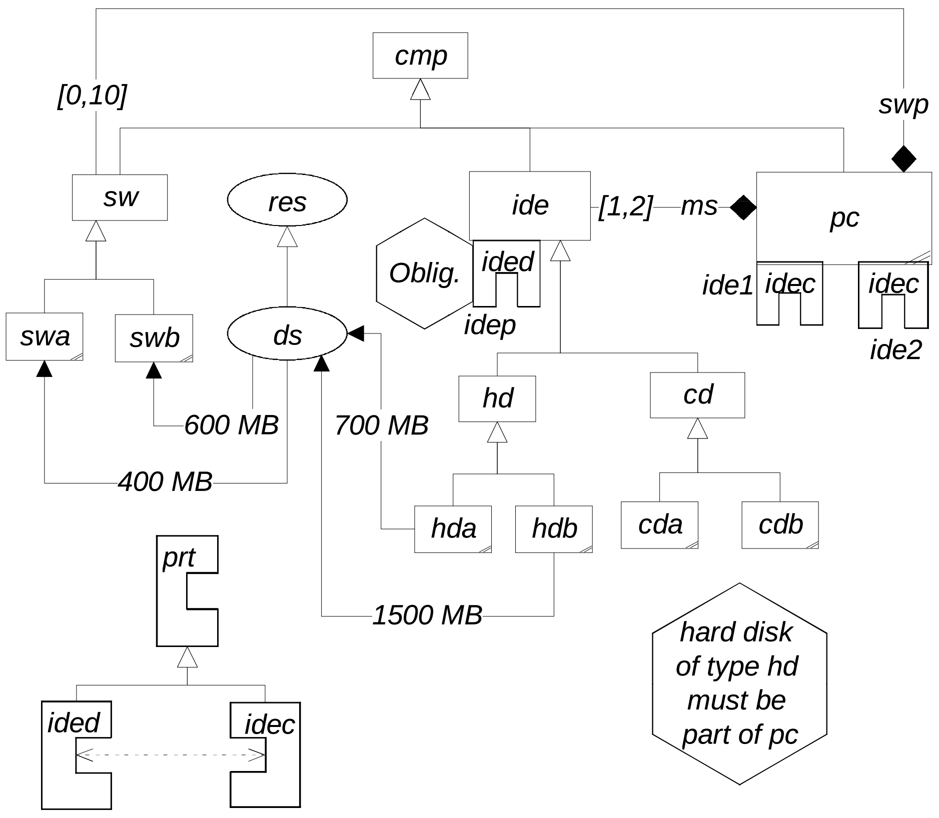


Figure 2.2 A PC configuration model

2.2.2 Compositional Structure

The decomposition of a product down 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. In **Figure 2.3**(a) a simple visual notation for representing the part definitions is given.

For simplicity, it is assumed that there is exactly one independent component type, referred to as a *root component type*. An individual of this type serves as the root of the compositional structure. There is exactly one individual of the root type in a configuration. Component types are assumed to be exclusive for simplicity's sake, meaning that a component individual is part of at most one component individual. Moreover, a component type or its super- or subtype may not occur 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 in a configuration is a tree of component individuals, and each component individual in a configuration is in the tree.

Example 2.2 The compositional structure of the PC is as follows (**Figure 2.2**). The root component type *pc* has as its parts 1 to 2 mass-storage units (with part name *ms*) of type *ide*, and 0 to 10 optional software packages (with part name *swp*) of the type *sw*.

■

2.2.3 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. In **Figure 2.3**(b) a simple visual notation for representing the production and use definitions is given.

Example 2.3 Disk space is used by the software packages and produced by hard disks. Disk space is represented by the resource type *ds* (**Figure 2.2**) which is a subtype of the generic resource type *res*. Each subtype of type *sw* uses a fixed amount of disk space, represented by their use definitions: *swa* uses 400MB and *swb* 600 MB. Hard disks of the type *hda* produce 700MB and of type *hdb* 1500MB of *ds*.

■

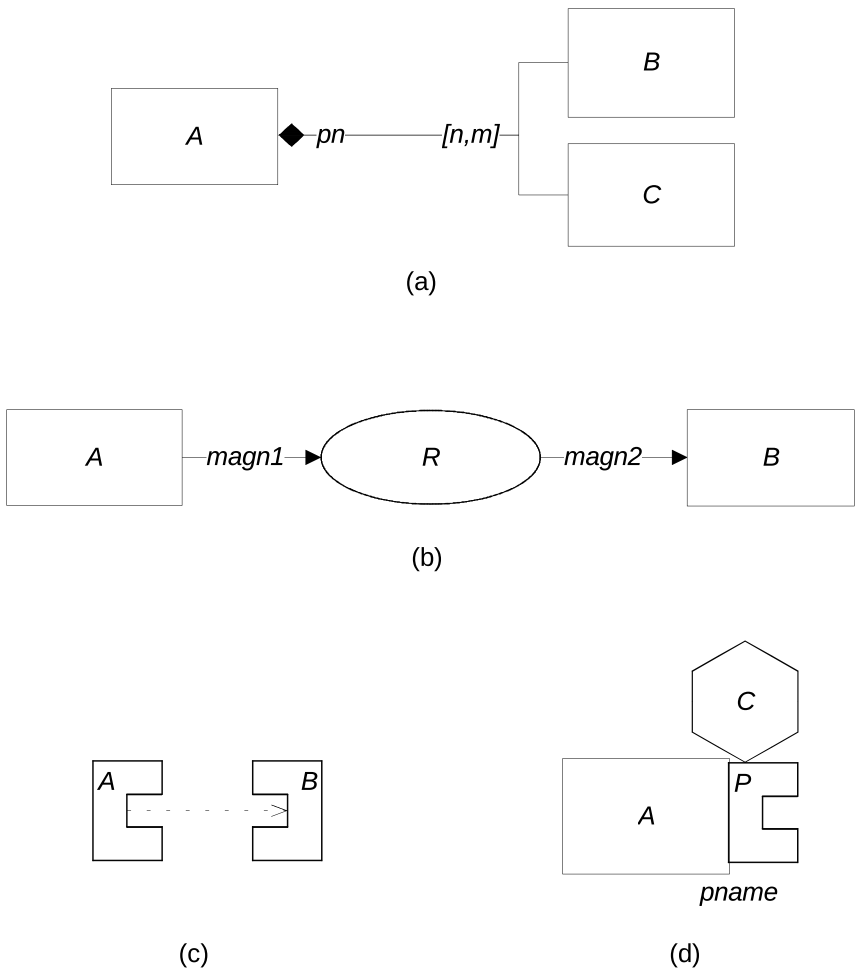


Figure 2.3 Notation for definitions: (a) Component type *A* has a part definition with part name *pn*, cardinality $[n, m]$ and two possible part types, *B* and *C*. (b) Component type *A* has a production definition with resource type *R*, and magnitude *magn1*. Component type *B* has a use definition with resource type *R*, and magnitude *magn2*. (c) Port type *A* is compatible with port type *B*. (d) Component type *A* has a port definition with port name *pname*, port type *P* and connection constraint *C*.

2.2.4 Ports and Connections

In addition to hierarchical decomposition, it is often necessary to model connections or compatibility between component individuals. The idea is that component individuals can be connected only if they have compatible interfaces. 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, only a limited connection constraint specifying whether a connection to a port individual is obligatory or optional is considered. In **Figure 2.3**(c) a simple visual notation for representing the compatibility and in **Figure 2.3**(d) a notation for port definitions are given.

Example 2.4 The configuration model includes port types *idec* and *ided*, subtypes of the general port type *prt* (**Figure 2.2**). These types represent the computer and peripheral device sides of the IDE connection. The compatibility definition of *idec* states that it is compatible with *ided*. Correspondingly, *ided* declares compatibility with *idec*.

The component type *pc* has two ports with the names *ide*1 and *ide*2 of the type *idec* for connecting IDE devices. The component type *ide* has one port of the type *ided*, called *idep*, for connecting the device to a computer. The *idep* port definition has the connection constraint that connection to that port is obligatory.

■

2.2.5 Constraints

All approaches to configuration knowledge representation have some kind of *constraints* as a mechanism for defining the conditions that a correct configuration must satisfy, i.e. the interdependencies between the types in the configuration model. A constraint is a formal rule, logical or mathematical or a mixture of these. In **Figure 2.1**(a), a simple way of embedding textual constraint descriptions in a visual notation for representing compatibility as well as other constraints is given.

Example 2.5 In the PC configuration model, there is the constraint that a hard disk of the type *hd* must be a part of *PC*. For now, it is given in natural language as a formal language has not yet been defined (**Figure 2.2**).

■

2.3 Configuration Rule Language

In this and the two following sections, attention is turned to the formal modeling of configuration knowledge. In this section, a simple formal rule-based Configuration Rule Language (**CRL)** for representing configuration knowledge is described (III). The aim is to define a simple language for representing abstracted, simplified configuration knowledge. The language extends those developed within the fields of logic programs and their non-monotonic semantics (Lloyd 1987; Przymusinska and Przymusinski 1990) by constructs useful in the configuration domain. The syntax and semantics of the language are informally presented and the formal, declarative semantics of the language briefly discussed. In addition, a configuration model of a simple configurable product is formalized using the language.

2.3.1 Intuition

There are several different languages for modeling configuration knowledge. A common denominator is that each of them defines some basic *elements* out of which a configuration is combined. These may be components, functions, statements about the properties of these, or the structure, connections or resource use of the product. In **CRL**, these are modeled uniformly as *atomic propositions* or *atoms* for short, with no commitment to their inner structure or information content. As an example, elements representing components in a personal computer domain could include different types of displays, mass memories such as hard disks, CD ROMs, floppy disks, RAM disks, and different types of extension cards.

A second common denominator is that there is a means for representing *choices* from a set of *alternative* elements. In **CRL**, choices among a set of elements are represented using *choice-rules*, allowing conditions for when choices are made, of the following two forms:

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

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

For brevity, the left-hand side of the rule is referred to as the *head*, and the right-hand side as the *body*, using the standard terminology of logic programs. Furthermore, the atoms without a preceding "not" are called positive literals and those with a preceding "not" negative literals. The meanings of the rules are: if elements $b_1,\ldots,b_m$ are in a configuration and elements $c_1,\ldots,c_n$ are not, then a choice between elements $a_1,\ldots,a_l$ must be made. In the case of rules of the first form, the choice is *inclusive*, i.e. any subset of the alternatives may be chosen. For rules of the latter form, the choice is *exclusive*, i.e. exactly one of the alternatives must be chosen. It is important to note that if the condition expressed by the body of a choice rule does not hold in a configuration, then the choice in that rule is not made. This means that in constructing a configuration, the set of choices changes dynamically according to the results of other choices. This aspect is captured by the semantics (III).

Example 2.6. Consider the following set of rules:

$$\begin{aligned} \mathit{computer} &\leftarrow \\ \mathit{IDEDisk} \mid \mathit{SCSIdisk} \mid \mathit{floppydrive} &\leftarrow \mathit{computer} \\ \mathit{FinnishlayoutKB} \oplus \mathit{UKlayoutKB} &\leftarrow \mathit{computer} \end{aligned}$$

The first rule states that there is a computer in the configuration. The second rule states that if there is a computer in a configuration, then one or more mass memory, which may be an IDE disk, SCSI disk or a floppy drive must be chosen. The third rule states that if there is a computer in the configuration, then either a keyboard with a Finnish layout or a keyboard with an UK layout must be chosen, but not both.

■

The third common denominator is that in addition to elements and choices between elements, there are *dependencies* between the elements. In **CRL**, the dependencies are expressed as either *requires*-rules or *incompatibility*-rules, expressed as follows, respectively:

$$\begin{aligned} a_1 &\leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n) \\ &\leftarrow b_1,\ldots,b_m,\mathrm{not}(c_1),\ldots,\mathrm{not}(c_n) \end{aligned}$$

The meaning of the requires-rule is that the presence of elements $b_1,\ldots,b_m$ in a configuration and the lack of elements $c_1,\ldots,c_n$ require the element a_1 to be in the configuration. Again, only if the condition expressed by the body of the rule holds in a configuration, the head element can be added to the configuration. The meaning of the incompatibility rule is that $b_1, \ldots, b_m$ being in a configuration and $c_1, \ldots, c_n$ not being in the configuration is not compatible, i.e. it must not be the case that $b_1, \ldots, b_m$ are in the configuration while $c_1, \ldots, c_n$ are not.

Example 2.7. As an example of dependencies in the PC domain, it could be that an SCSI hard disk requires a specific SCSI controller, and that a PII processor does not work with the I820 motherboard. This would be represented by the rules

$$\begin{aligned} SCSIcontroller &\leftarrow SCSIdisk \\ &\leftarrow PIIprocessor, I820motherboard \end{aligned}$$

■

2.3.2 Syntax and Semantics

As requires and incompatibility rules are special cases of both inclusive and exclusive choice rules, the above types of knowledge are represented uniformly in **CRL** using rules of the form

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

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

where a_i, b_i, c_i are propositional atoms.

In **CRL** semantics, a configuration is a set of atoms, a configuration model a set of rules and a set of requirements also a set of rules. The declarative semantics of **CRL** is based on the model theoretic semantics of propositional logic, by interpreting "," as conjunction, "not" as negation, "$|$" as disjunction, "$\oplus$" as exclusive disjunction, and "$\leftarrow$" as implication. A configuration *satisfies* a set of rules if the corresponding set of atoms satisfies each rule. A configuration satisfies a rule if, when its body is satisfied by a configuration, then its head is also satisfied by the configuration. In **CRL** terms, a correct configuration with respect to a configuration model and a set of requirements must satisfy the two sets of rules in the configuration model and the requirements. However, this is not a sufficient condition.

The semantics of **CRL** additionally includes the property of *groundedness* or *supportedness*. Intuitively, groundedness means that there must not be any element in the configuration that is not necessary to satisfy the rules in the configuration model. This property guarantees that a configuration such as $\{computer, IDEdisk, UKlayoutKB, SCSIcontroller, MPEGcard\}$ is not correct with respect to the PC configuration model presented in the previous section and the empty set of requirements. This is due to two reasons. First, the SCSI controller is not needed for the configuration to be correct, since there is no SCSI disk in the configuration. Second, there is no rule in the configuration model implying that an MPEG card can be included in the configuration.

Requiring groundedness changes the semantics of the language from that of standard propositional logic, particularly the meaning of the connectives of "←" and "not". The "not"-connective has a default semantics, which roughly means that a negated literal is true if the atom within it cannot be derived using the rules. This contrasts with a classically negated literal, which is true only if the negative literal itself can be derived from the rules when considering the "←" connective as an inference rule.

The groundedness property is formalized as follows: the presence of each element in a configuration must be grounded by being derivable using the rules in the configuration model. That is, every element, i.e. atom, in a configuration must occur in the head of at least one rule, whose body consists of, a) positive literals that are in a configuration and are in turn grounded similarly, and b) negative literals that are not in a configuration.

This property is formally defined using a technique similar to that used in the semantics of logic programs. It is expressed as a fix-point equation on configurations. The condition states that after a transformation depending on the configuration that removes the negative literals from the rules in the configuration model the following must hold: forward chaining of the transformed rules must produce exactly the original configuration. If this is the case and the configuration satisfies the rules in the configuration model, the configuration is *correct*. If the configuration further satisfies the requirements, it is *suitable*. Such a configuration, i.e. a set of atoms, is then called a *stable model* of the set of rules.

2.3.3 Example

In this section the use of **CRL** is demonstrated by representing the configuration knowledge of a simple PC configuration problem.

Example 2.8. A computer is configured using the following configuration model: one must choose

- a mass-memory, one or more of the set $\{IDEdisk, SCSIdisk, floppydrive\}$,
- a keyboard, exactly one of the set $\{FinnishlayoutKB, UKlayoutKB\}$,
- a processor, exactly one of the set $\{PII, PIII\}$,
- a motherboard, one of the set $\{ATX, I820\}$, and
- a graphics card, $gcard$.

In addition, the following dependencies must be respected:

- the SCSI disk requires an SCSI controller ($SCSIcontroller$),
- the *PII* processor is incompatible with the *I820* motherboard and the *PIII* processor with the *ATX* motherboard, and
- a graphics card need only be chosen if the motherboard does not contain one embedded in it. An *ATX* motherboard contains one, but an *I820* does not.

The following rule set represents this configuration model:

$$\begin{aligned}
computer &\leftarrow \\
IDEDisk \mid SCSIdisk \mid floppydrive &\leftarrow computer \\
FinnishlayoutKB \oplus UKlayoutKB &\leftarrow computer \\
PII \oplus PIII &\leftarrow computer \\
ATX \oplus I820 &\leftarrow computer \\
SCSIcontroller &\leftarrow SCSIdisk \\
&\leftarrow PII, I820 \\
&\leftarrow PIII, ATX \\
gcard &\leftarrow not(gcardinmb) \\
gcardinmb &\leftarrow ATX
\end{aligned}$$

Given this configuration model, consider the requirements $\{FinnishlayoutKB \leftarrow\}$ and the following configurations:

$$C_1 = \{computer, SCSIdisk, UKlayoutKB, PII, PIII, gcard\}$$

$$C_2 = \begin{Bmatrix} computer, IDEDisk, FinnishlayoutKB, PIII, ATX, SCSIcontroller, \\ gcardinmb \end{Bmatrix}$$

$$C_3 = \begin{Bmatrix} computer, SCSIdisk, FinnishlayoutKB, PII, ATX, SCSIcontroller, \\ gcardinmb \end{Bmatrix}$$

The configuration C_1 is not correct for several reasons. It does not satisfy the configuration model, since there are two processors in the configuration, which conflicts with the exclusive choice in the fourth rule of the configuration model. In addition, no motherboard has been chosen as per the fifth rule although a computer is in the configuration. The dependency that if there is no graphics card in the motherboard then one must be added is satisfied, though. Configuration C_1 does not satisfy the requirements either, since there is no Finnish layout keyboard in the configuration.

The configuration C_2 does satisfy the requirements. However, it does not satisfy the configuration model, since there is a PIII processor and an *ATX* motherboard in it, an incompatible combination according to the eighth rule in the configuration model. Even though this problem were fixed by changing the processor or motherboard, the changed C_2 still would not be correct, as it is not valid. This is due to the presence of the SCSI controller in the configuration, although no rule supports its inclusion. By changing the processor and the type of mass-memory, the configuration C_3 is obtained. This configuration satisfies the configuration model and the requirements. Furthermore, it is now correct, since the SCSI disk now provides the reason for including the SCSI controller in the configuration.

■

2.4 Extended Dynamic Constraint Satisfaction Problem

Dynamic constraint satisfaction problems (DCSP) are a formalism introduced in (Mittal and Falkenhainer 1990) for capturing the dynamic nature of configuration knowledge. Where **CRL** introduced in the previous section extends logic programs, DCSP extends constraint satisfaction problems (CSP) (Mackworth 1977), another popular formalism, to incorporate configuration specific features. In this section, the use of CSP in representing configuration knowledge is briefly explained. The original notion of DCSP is introduced and motivated by the dynamic nature of configuration problems. Then DCSP is extended with more expressive constructs motivated by the configuration domain. The intuition behind a new declarative semantics for the extended DCSP (EDCSP) that coincides with the original when restricted to the original DCSP class is briefly outlined. The semantics uses a fix-point condition similar to the one discussed in the previous section instead of the original minimality condition introduced in (Mittal and Falkenhainer 1990). This allows a computationally more feasible semantics (II).

2.4.1 Constraint Satisfaction Problems Applied to Configuration

A CSP consists of a set of *variables*, a set of possible *values* for each variable, called the *domain* of the variable, and a set of *constraints* (Mackworth 1977). A constraint defines the allowed combinations of values for a set of variables by specifying the allowed subset of the Cartesian product of the domains of the variables. A *solution* to a CSP is an *assignment* of values to all variables such that the constraints are satisfied, i.e., the value combinations are allowed by at least one tuple of each constraint.

When applying CSP to configuration knowledge presentation, a configuration model becomes an instance of CSP. A variable in the CSP instance represents a choice and the domains of the variables represent the possible alternatives for the choices. The constraints are used to represent the dependencies between choices. A solution to the CSP instance corresponds to a configuration. Requirements are represented as an additional set of constraints that a correct configuration must satisfy.

Example 2.9. A simplified configuration task for an industrial mixer (II) is used to illustrate CSP and DCSP in this section. An industrial mixer can be used for different mixing processes, such as chemical reactions, the mixing of side products etc. It consists of a set of standard components, such as a vessel containing the products to be mixed, the mixer itself with impellers and an engine. Depending on the chemical properties of the products to be mixed, heat is produced which requires the use of a cooler or a condenser. To represent the configuration model of a mixer as a CSP, the components and their properties, for example the volume, are represented as variables. The mixing process is represented by a variable for the mixing task. The components can be of different types; e.g. the mixer can be a reactor, storage tank, or simple mixer. These as well as the different types of mixing tasks and the volume of the vessel are represented as discrete domains of the variables. To designate variables and values in the problem, the first letters of their names are used. A partial configuration model for the mixer configuration task consists of the

Table 2.1 Variables and their domains in the mixer configuration model.

Variable	Description	Domain
Mt	Mixing task	$\{d(ispersion), s(uspension), b(lending)\}$
Mi	Mixer	$\{m(ixer), r(eactor), t(ank)\}$
Coo	Cooler	$\{cool(cooler1)\}$
Con	Condenser	$\{con1(condenser1), con2(condenser2\}$
Vol	Volume	$\{l(arge), s(mall)\}$

variables in **Table 2.1** and the following constraints on condenser and volume and mixer and volume, respectively:

$$c_1(Con, Vol) = \{(con1, l), (con2, l), (con2, s)\}$$
$$c_2(Mi, Vol) = \{(r, s), (m, s), (m, l), (t, s), (t, l)\}$$

Given the above mixer CSP, and the requirement that the mixing task is of the dispersion type, represented as a constraint $c_3(Mt) = \{(d)\}$, the assignment $\{Mt = d, Mi = m, Coo = cool, Con = con1, Vol = l\}$ is a correct configuration as it is a solution of the CSP, i.e. it satisfies the constraints in the configuration model, and satisfies the requirement. The assignment $\{Mt = d, Mi = m, Coo = cool, Con = con1, Vol = s\}$ is not a correct configuration, since, although it satisfies the requirement, it does not satisfy c_1.

■

2.4.2 Dynamic Constraint Satisfaction Problems

Product configuration problems exhibit dynamic aspects in the generation of problem spaces. This means that the set of choices, i.e. variables, in a solution changes dynamically based on other choices, i.e. assignments of values to variables. When configuring a mixer, for example, a condenser is a typical optional component that does not have to be present in every solution. It is only necessary if the vessel volume is large and chemical reactions occur during the mixing process.

Such dynamic aspects are difficult to capture in a standard CSP in which all variables are assigned values in every solution. One way to deal with an optional component is to add a special NULL value to the domain of the variable representing the component (Gelle 1998). In addition, each constraint that refers to the variable needs to be modified to function properly in the presence of NULL. More seriously, an additional constraint is needed to prevent values other than NULL for the variable if there is no reason for including the optional component in the solution. Such constraints may have a very large arity and cardinality, as many variables may affect the value of the variable representing the optional component. Much effort has therefore been invested to include dynamic aspects in CSPs (Mittal and Falkenhainer 1990; Stumptner et al. 1998; Sabin and Freuder 1996; Gelle 1998). One of these is the framework of dynamic constraint satisfaction problems CSPs (Mittal and Falkenhainer 1990) which adds *activity constraints* to CSP. The activity constraints govern which variables are given values, i.e. are *active*, in a solution.

A DCSP is an extension of a CSP that also has of a set of variables, domains and constraints (called here *compatibility constraints*). However, all the variables need not be given a value, i.e., be in a solution. Hence, a compatibility constraint is defined to be satisfied by an assignment if the variables it constrains are active in the assignment and their assigned values are allowed by the constraint, or at least one of the variables is not active in the assignment.

A DCSP additionally defines a set of *initial variables* that must be active in every solution and a set of *activity constraints*. An activity constraint states either that if a given condition is true then a certain variable is active, or that if a given condition is true, then a certain variable must not be active. The condition may be expressed as a set of compatibility constraints (*require* and *require not* activity constraints) that must all be satisfied and *active*, i.e., the variables they refer to must all be active. Alternatively, a condition may state that some set of other variables are active (*always require* and *always require not* activity constraints). An activity constraint is satisfied by an assignment if 1) its condition is not active, i.e. at least one of the constraints or variables in the condition is not active in the assignment, or if 2) the condition is active and the activated variable is active in the assignment.

A solution to a DCSP is an assignment of values to variables such that it 1) satisfies the compatibility and activity constraints, 2) contains assignments for the initial variables, and 3) is subset minimal. The subset minimality condition in effect excludes assignments with active variables whose activity is not justified by the variables being in the set of initial variables or by at least one activity constraint which is satisfied and active.

Example 2.10. The configuration model of the industrial mixer presented in **Example 2.9** also includes dynamic aspects, which will be now addressed. The constraints in that example are now compatibility constraints.

A mixing task, mixer type and volume need to be chosen in every configuration of a mixer. This is represented by including the corresponding variables in the set of initial variables, i.e. $V_I = \{Mt, Mi, Vol\}$. A cooler and a condenser are not needed in every solution, but only if the mixer is of the reactor type and the mixing task is dispersion, respectively. Therefore they are not included in the set of initial variables but are introduced by the activity constraints a_1 and a_2:

$$a_1 = \left(Mi = r \overset{ACT}{\rightarrow} Coo \right)$$

$$a_2 = \left(Mt = d \overset{ACT}{\rightarrow} Con \right)$$

An extended configuration model for the mixer configuration task represented as a DCSP consists of the variables and compatibility constraints in **Example 2.9** and the set of initial variables and activity constraints defined here.

Given the above mixer DCSP, the assignment $\{Mt = b, Mi = m, Vol = l\}$ is a correct configuration as it is a solution of the DCSP, i.e. satisfies the activity and compatibility constraints in the configuration model, contains assignments for the initial variables, and there is no other solution which would be its subset. The assignment $\{Mt = b, Mi = r, Coo = coo1, Vol = s\}$ is another correct configuration with a different set of active variables. However, the assignment $\{Mt = b, Mi = r, Coo = coo1, Con = con2, Vol = s\}$ is not a correct configuration, since the second configuration is its subset, and therefore the initial variables or activity constraints do not justify the assignment of a condenser.

■

2.4.3 Extending Dynamic Constraint Satisfaction Problems

The activity constraints of DCSP are not particularly expressive. For instance, in some configuration tasks a functional requirement can be satisfied by any of a given set of components, which would require *disjunctive activity constraints* (Mittal and Falkenhainer 1990). Another case that cannot be represented in a straightforward manner is that a choice needs to be made, i.e., a variable is active, under the condition that some variables are not active or not given particular values in an assignment. In other words, it would also be useful to allow conditions referring to the complement of activity or constraints.

The DCSP is thus extended to *extended dynamic constraint satisfaction problem* (EDCSP) by allowing *generalized activity constraints* of the following form:

$$c_1,\ldots,c_j, not(c_{j+1}),\ldots, not(c_k) \overset{ACT}{\rightarrow} m\{v_1 \mid \cdots \mid v_l\}n$$

where $0 \le m \le n$. Intuitively, this activity constraint states that if constraints $c_1,\ldots,c_j$ are active in and satisfied by an assignment and the constraints $c_{j+11},\ldots,c_k$ are not satisfied or not active in the assignment, then a subset of the variables $v_1,\ldots,v_l$ is activated, such that the cardinality of the subset is between m and n. If $m=1$ and $n=l$ this becomes an inclusive disjunction or choice of the variables. On the other hand, if $m=n=1$, the right hand side becomes an exclusive disjunction or choice. Rules with $m=0$ are also allowed for representing *optional* variables, i.e., variables that may be active or inactive, and rules with $n=0$ for expressing 'requires not' activity constraints. 'Always require' and 'always require not' activity constraints, i.e. activity constraints where the constraints are replaced by variables are again treated as short hand notation for constraints that allow any value of the variable.

Example 2.11. In **Example 2.10**, the configuration $\{Mt=d, Mi=r, Coo=coo1, Con=con2, Vol=s\}$ with a condenser and a cooler is correct. In order to avoid having both condenser and cooler active, consider the following DCSP obtained by replacing the activity constraint a_1 by an "exclusively disjunctive" activity constraint:

$$a_3 = \left(c_4 \overset{ACT}{\rightarrow} 1\{Coo \mid Con\}1 \right)$$

$$c_4(Mi, Mt) = \{(r,d)\}$$

Now, the assignment $A_1 = \{Mt=d, Mi=r, Con=con2, Vol=s\}$ is a solution since it clearly satisfies the constraints, the initial variables are active in it and all the active variables are justified by activity constraints whose condition is active. The activity constraint a_4 is satisfied, since its condition is active and satisfied and exactly one of the activated variables Coo and Con is active, as required by the cardinality limits. On the other hand, $\{Mt=d, Mi=r, Con=con2, Coo=coo1, Vol=s\}$ is not a solution, since it does not satisfy the cardinality limits in a_3.

■

Example 2.12 If the activity constraints of **Example 2.10** were replaced with the activity constraint

$$a_4 = \left(not(Coo = cool1) \overset{ACT}{\rightarrow} 1\{Con\}1 \right)$$

with default negation, the assignment A_1 would still remain a solution, since a_4 is satisfied in it and the condenser variable is active. The condition of a_4 is satisfied, since the cooler variable is not active.

■

Example 2.13 If the activity constraints of **Example 2.10** were instead replaced with the "inclusively disjunctive" activity constraint

$$a_5 = \left(c_5 \overset{ACT}{\rightarrow} 1\{Coo \mid Con\}2 \right)$$
$$c_5(Mi, Mt) = \{(r, d)\}$$

the assignment A_1 would still remain a solution, since a_5 is satisfied in it and activates the condenser variable. However, now also the assignment $\{Mt = d, Mi = r, Con = con2, Coo = cool1, Vol = s\}$ is a solution, since the cardinality limits of a_5 allow it to activate two variables.

■

Extending the original definition of the semantics of DCSP with such constraints would easily lead to highly complex computational problems (II). Therefore, in (II) a novel semantics for the original and extended DCSP is developed whose computational complexity remains lower. In the semantics, the minimality condition is replaced with a transformation of the activity constraints with respect to a potential solution, and a fixpoint definition based on an operator on the lattice of solutions. This semantics is similar to the semantics of **CRL** in that it preserves a form of groundedness. The active variables in a solution must be grounded by them being in the set of initial variables or by activity constraints, the variables of whose left hand side are also similarly recursively justified. In the definition, the minimality requirement is in fact relaxed. The activity constraints or initial variables justify each active variable in a solution, but a disjunctive activity constraint may justify more variables than a subset minimal solution would contain, as demonstrated by **Example 2.13**.

2.5 Weight Constraint Rule Language

In this section a first order weight constraint rule language (**WCRL**) is described (I). The language includes cardinality constraints for representing complex choices and weight constraints for representing resource constraints. It is first-order, i.e. includes predicates and variables, for representing rules and relations between different types of objects in a configuration in a compact manner. The languages for representing configuration knowledge presented in previous sections do not offer adequate support for representing and reasoning on these aspects. The syntax and declarative semantics of **WCRL** are informally presented and an example of its use in representing configuration knowledge is given.

2.5.1 Cardinality Constraints

A natural extension of the inclusive and exclusive choices of **CRL** (Section 2.1) is to allow *cardinality constraints* for representing choices where the number of alternatives chosen is more generally restricted. These choices are similar to the cardinality bounds in the activity constraints of the extended DCSP (Section 2.4) and can be used in place of choices in rules. A rule with a cardinality constraint in its head is an expression of the following form:

$$L \leq \{a_1, \ldots, a_n\} \leq U \leftarrow b_1, \ldots, b_l, not(c_1), \ldots, not(c_m)$$

The intuitive meaning of such rules is that if $b_1, \ldots, b_l$ are in a configuration and $c_1, \ldots, c_m$ are not, then a subset of $a_1, \ldots, a_n$ such that the cardinality of the subset is between L and U (lower and upper bound, respectively), including L and U, is in a configuration.

Example 2.14 Consider a situation where there are three slots for extension cards in a PC, and a selection of five different cards that can be connected to the slots, one per slot. In addition, only one of each card can be in the configuration. However, the slots can be empty. The choice of 0 to 3 cards out of 5 possibilities without repetition of the same card can be represented by the following rule:

$$computer \leftarrow$$

$$0 \leq \{card_1, card_2, card_3, card_4, card_5\} \leq 3 \leftarrow computer$$

The configuration $\{computer\}$ satisfies these rules, as well as the configuration $\{computer, card_1, card_3, card_5\}$, since the cardinality limits of the weight constraint are satisfied. However, the configuration $\{computer, card_1, card_3, card_5, card_4\}$ is not valid, since the upper bound of the weight constraint is not satisfied.

■

2.5.2 Weight Constraints

For some products, it is natural to model the dependencies between components as resource production and use. Sometimes it is also necessary to impose global constraints on some resource usage, such as the total cost of the configuration. Representing such knowledge motivates generalizing cardinality constraints with *weights*. The idea is to view the set of atoms in a cardinality constraint as a sum over those atoms satisfied by a configuration, whose value must be between the lower and upper bound. In order to represent resource constraints, a real-valued unique coefficient, weight, is associated with each atom, i.e. summand. In addition, an atom must not be repeated in such a weight constraint. A rule with a weight constraint in its head is of the following form:

$$L \leq \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}\} \leq U \leftarrow b_1, \ldots, b_l, not(c_1), \ldots, not(c_m)$$

The intuitive meaning of such rules is that if $b_1, \ldots, b_l$ are in a configuration and $c_1, \ldots, c_m$ are not, then the sum of the weights of those atoms of $a_1, \ldots, a_n$ that are in a configuration must be between L and U, inclusive, i.e. the following inequality must hold:

$$L \leq \sum_{\substack{a_i \text{ is in} \\ \text{configuration,} \\ 1 \leq i \leq n}} w_{a_i} \leq U$$

Example 2.15 Consider again the situation where there is a selection of 5 different cards that can be connected to the slots in the motherboard. However, the cards consume power in different amounts, and the power source of the PC is only capable of producing 100W. The production and consumption of the power resource and the natural dependency that the production of power must be at least as much as its consumption can be captured using the following rules:

$$computer \leftarrow$$

$$powersource \leftarrow$$

$$0 \le \left\{ \begin{array}{l} card_1 = -20, card_2 = -20, card_3 = -40, \\ card_4 = -40, card_5 = -80, powersource = 100 \end{array} \right\} \le 100 \leftarrow computer$$

The configuration $\{computer, powersource\}$ satisfies these rules, as well as the configuration $\{computer, powersource, card_1, card_2, card_4\}$, since the sum of the weights of the atoms in the configuration and in the weight constraint is 100 in the first case and 20 in the second case, thus greater than the lower bound.

■

2.5.3 Propositional Weight Constraint Rule Language

Cardinality constraints can be viewed as a special case of weight constraints with each literal having weight one. Therefore, the extension of **CRL** is based on weight constraints and the resulting language called propositional weight constraint rule language (**PWCRL**). Weight constraints are also useful in representing disallowed situations in the body of an incompatibility rule. Furthermore, it is useful to allow negated literals in addition to positive ones within the body weight constraints. Therefore, the rules of **PWCRL** are of the following uniform form:

$$C_0 \leftarrow C_1, \ldots, C_n$$

where $C_0, \ldots, C_n$ are weight constraints of the form

$$L \le \{a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, not\ a_{n+1} = w_{a_{n+1}}, \ldots, not\ a_m = w_{a_m}\} \le U$$

and C_0 contains only positive literals. The intuitive meaning of such rules is that if the weight constraints in the body of the rule are satisfied in a configuration, then the weight constraint in the head of the rule must also be satisfied in the configuration. The notion of when a weight constraint is satisfied must be extended to handle negated literals as follows:

$$L \le \sum_{\substack{a_i \text{ is in} \\ \text{configuration,} \\ 1 \le i \le n}} w_{a_i} + \sum_{\substack{a_i \text{ is not in} \\ \text{configuration,} \\ n+1 \le i \le m}} w_{a_i} \le U$$

In addition, any literal must not be satisfied in a configuration unless it is grounded by appearing in the head constraint of at least one rule whose every body constraint is satisfied by literals recursively grounded by other rules. In addition to "$\leq$", strict inequality can be introduced similarly and can be used in the rules as well.

Such rules are a generalization of the choice- and requires-rules of **CRL**, as easily can be seen. An inclusive choice rule

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

can be represented using a weight constraint rule by noting that a literal l (negated or positive) can be seen as a shorthand for the *simple weight constraint* $1 \leq \{l = 1\} \leq 1$, and the choice captured using a weight constraint, as follows:

$$\begin{aligned} 1 \leq \{a_1 = 1, \ldots, a_l = 1\} \leq l \leftarrow 1 \leq \{b_1 = 1\} \leq 1, \ldots, 1 \leq \{b_m = 1\} \leq 1, \\ 1 \leq \{not\ c_1 = 1\} \leq 1, \ldots, 1 \leq \{not\ c_n = 1\} \leq 1 \end{aligned}$$

An exclusive choice rule can be represented similarly by setting the upper bound to one. The *head* constraint C_0 may be missing, in which case the rule becomes an incompatibility rule, i.e. that at least one of the *body* constraints $C_1, \ldots, C_n$ must not be satisfied. For brevity, the above short hand notation for literals, i.e. l stands for $1 \leq \{l = 1\} \leq 1$, is often used. Moreover, the weights in a weight constraint are omitted if they all are 1 and the bounds integers, i.e. if the constraint is a cardinality constraint. In this case, the relation operators can also be omitted. Another convention is that the lower or upper bound of a weight constraint may be omitted, in which case it is taken to be $-\infty$ or ∞, respectively.

2.5.4 First-order Weight Constraint Rule Language

So far in the treatment of the languages for configuration knowledge representation in this thesis, the actual information content of the elements, i.e. atoms, in a configuration has been omitted. This aspect is tackled in this section by generalizing **PWCRL** by replacing propositional atoms with atoms constructed of predicates, variables and object constants, following the usual procedure for first-order languages (Hodges 1983). This also facilitates a much more compact representation of dependencies between sets of objects in a configuration using rules. The notion of conditional atoms is also added to support the representation of knowledge pertaining to only certain types of objects. The resulting language is called the first-order weight constraint rule language (**FOWCRL)**. However, the whole machinery required for first-order languages is not dealt with here. (Hodges 1983) gives a thorough exposition of these issues.

In representing configuration, and indeed any, knowledge, it is useful to be able to refer to the objects that may appear in a configuration, all or some objects in the domain, and the relations between the objects. The method for accomplishing this is to introduce *object constant*, *variable* and *quantifier*, and *predicate* symbols into the language.[4] Informally, an object constant symbol refers to an object in the domain that knowledge is being represented on. A variable symbol refers to some object in the domain. A quantifier is associated with each variable to express knowledge such as that "all" or "some" of the objects in the domain have a given property or are related to each other in a given way. Predicate symbols refer to these properties of and relations between objects.

The change from the propositional rule language to a first-order one is syntactically accomplished by first replacing atomic propositions with *atoms*[5] of form

$$p(x_1,\ldots,x_n)$$

where p is an *n-ary* predicate symbol and $x_1,\ldots,x_n$ are either object constant symbols or variable symbols, collectively referred to as *terms*. Intuitively, such an atom expresses the situation where $x_1,\ldots,x_n$ are related to each other via the relation named by p. In writing the rules, the standard notation of using uppercase first letters for variables and lower case letters for constant object and predicate symbols is adopted.

Note that an atomic proposition can be considered a *0-ary* atom, i.e. an atom without any terms. In addition, an atom without any variables, i.e. only constants, is also equivalent to an atomic proposition. Whether an atomic proposition is satisfied, i.e. true, in a configuration can be defined by referring to the atomic proposition only. In contrast, the truth-value of an atom with arity greater than zero and with variables depends on what values its component variables assume. For example, the relation *less than*, denoted by the two-ary predicate "$<(x,y)$", is true if x refers to 1 and y to 2, i.e. $<(1,2)$ is true, but not for 2 and 1, in the usual natural number interpretation of the object constants 1 and 2.

[4] Function symbols are omitted for reasons discussed in Section 4.3.2.

[5] Hereafter, the term atom refers to the construct defined here, and when referring to an atom in the propositional sense, the term atomic proposition is used.

The truth-value of an atom with a variable such as $<(x,2)$ depends on what the variables, in this case x, refers to. Quantifiers "all" and "some", denoted by $\forall$ and $\exists$ and called the *universal* and *existential* quantifier, respectively, can also be attached to the variables. They express the knowledge that for all objects in the domain an atom is true or that for at least one object in the domain the atom is true, respectively.

The rules of **FOWCRL** are rules of **PWCRL** where propositions are replaced by atoms of form given above, and furthermore, each variable is universally quantified, i.e. of form:

$$\forall\vec{x}(C_0 \leftarrow C_1,\ldots,C_n)$$

where $\forall\vec{x}$ denotes universal quantification for all the variables in the rule and is usually assumed implicitly and omitted from the rules for brevity. C_0 may again be lacking. The weight constraints $C_1,\ldots,C_n$ are of the form:

$$L \le \begin{Bmatrix} p_1(\vec{x}_1) = w_{p_1},\ldots, p_n(\vec{x}_n) = w_{p_n}, \\ not\ p_{n+1}(\vec{x}_{n+1}) = w_{p_{n+1}},\ldots, not\ p_m(\vec{x}_m) = w_{p_m} \end{Bmatrix} \le U$$

where $\vec{x}_1,\ldots\vec{x}_m$ are the variables in the atoms $p_1(\ ),\ldots,p_m(\ )$. C_0 is similar to $C_1,\ldots,C_n$ except that it may not contain negative literals. The notational convention of logic programming, i.e. denoting object and predicate constants by names starting with lower case letters, and variables by names starting with upper case letters is followed when writing the rules.

Intuitively, the difference of first order rules to the propositional rules is that a first order rule is satisfied only if it is satisfied for every object in the domain. In other words, every propositional rule obtained by substituting object constants consistently (i.e. that the same variable is substituted with the same object everywhere in the rule) in place of variables in every possible way must be satisfied in a configuration for the first-order rule to be satisfied.

Example 2.16 Consider again the situation where there are 3 slots for extension cards in a PC, and a selection of 5 different card types that can be connected to the slots, one per slot. However, this time several cards of the same type are allowed in the configuration. Assume further that the allocation of cards to slots is an important part of the configuration task for installation purposes. The following rules using predicates and variables capture this configuration knowledge:

$$\begin{array}{r}
slot(s_1) \leftarrow \quad slot(s_2) \leftarrow \quad slot(s_3) \leftarrow \\
card(c_1) \leftarrow \quad card(c_2) \leftarrow \quad card(c_3) \leftarrow \\
card(c_4) \leftarrow \quad card(c_5) \leftarrow \qquad\qquad\quad \\
0 \le \{conn(s_1, C), conn(s_2, C), conn(s_3, C)\} \le 3 \leftarrow card(C) \\
\leftarrow slot(S), card(C_1), card(C_2), \\
conn(S, C_1), conn(S, C_2), \\
not\, eq(C_1, C_2) \\
eq(C, C) \leftarrow card(C)
\end{array}$$

The first eight rules introduce the three slots and five card types as objects in the domain. In addition, they introduce the two unary predicates $slot(\ )$ and $card(\ )$ whose extensions define the slot and card objects. The ninth rule captures the fact that all cards (C is universally quantified) can be connected to from zero to three slots. Here the predicate $conn(\ )$ on a slot and card is used to represent the fact that the slot is connected to the card. The tenth rule states that for each slot, there is at most one card connected to it, by requiring that if there were two cards connected to the slot, they would have to be the same card. The equality of two cards represented as the predicate $eq(\ )$ is defined simply to mean that the two are the same card object.

To see how the first order rules differ from propositional ones, consider the meaning of the last rule. It states that for all cards it holds that a card is equal to itself. In other words, the last rule is satisfied in the configuration only if the following rule set is satisfied (if the object constants in the rule set represent all the objects in the domain):

$$\begin{array}{c}
eq(s_1, s_1) \leftarrow card(s_1) \\
eq(s_2, s_2) \leftarrow card(s_2) \\
eq(s_3, s_3) \leftarrow card(s_3) \\
eq(c_1, c_1) \leftarrow card(c_1) \\
eq(c_2, c_2) \leftarrow card(c_2) \\
eq(c_3, c_3) \leftarrow card(c_3) \\
eq(c_4, c_4) \leftarrow card(c_4) \\
eq(c_5, c_5) \leftarrow card(c_5)
\end{array}$$

One could represent this configuration knowledge using propositional rules by introducing a distinct proposition $conn(card, slot)$ for each card-slot pair, 15 in all. However, instead of the ninth rule, one would have five similar rules, one for each card. The tenth rule is even more problematic, since to have the same effect the following 30 propositional rules are needed:

$$\leftarrow conn(S, C_1), conn(S, C_2),$$
$$\text{for } S \in \{s_1, s_2, s_3\}, C_1 \in \{c_1, c_2, c_3, c_4, c_5\}, C_2 \in \{c_1, c_2, c_3, c_4, c_5\}, C_1 < C_2$$

■

As demonstrated in **Example 2.16** universally quantified variables make it easier to capture knowledge compactly. However, a similar mechanism would be useful for capturing compactly the set of literals in a choice rule. Next, a mechanism for accomplishing this in **FOWCRL** is defined. It also allows expressing that a predicate ranges over a subset of the objects of domain only. This is useful in representing knowledge pertaining to certain classes or types of objects in a configuration model. After this extension, the entire **FOWCRL** has been introduced.

The mechanism is to allow the attachment of a *conditional part* to an atom, and to allow *conditional literals*, i.e. atoms or their negations with a conditional part, in place of previously defined atoms in rules. The resulting atom is called a *conditional atom* and is of the form

$$p(\vec{x}) : q(\vec{y})$$

where $p(\vec{x})$ and $q(\vec{y})$ are both atoms, called the *proper part* and *conditional part*, respectively. A conditional atom is satisfied in a configuration if both $p(\vec{x})$ and $q(\vec{y})$ are satisfied in a configuration. Conjunctions of arbitrary length are allowed in the conditional part as well, such as

$$p(\vec{x}) : q(\vec{y}) : r(\vec{z})$$

The idea is to use such atoms to limit the domains of variables in $p(\vec{x})$ to the objects for which $q(\vec{y})$ and $r(\vec{z})$ hold. For instance, the ninth rule in **Example 2.16** can be captured more compactly as:

$$0 \leq \{conn(S, C) : slot(S)\} \leq 3 \leftarrow card(C)$$

The intuition behind such a rule is that the set of literals in a weight constraint consists effectively of the proper parts of conditional literals for which the conditional part is satisfied in a configuration. More precisely, the variables that occur in several weight constraints such as C in the above rule are called *global variables*. They are defined, as earlier, to be universally quantified. The variables occurring in only one conditional literal are called *local variables*. The general definition of a first-order weight constraint rules remains the same, but the generalized weight constraints are now of the form:

$$L \leq \left\{ \begin{matrix} p_1(\vec{x}_1) : q(\vec{y}_1) = w_{p_1}, \ldots, p_n(\vec{x}_n) : q(\vec{y}_n) = w_{p_n}, \\ not\ p_{n+1}(\vec{x}_{n+1}) : q(\vec{y}_{n+1}) = w_{p_{n+1}}, \ldots, not\ p_m(\vec{x}_m) : q(\vec{y}_m) = w_{p_m} \end{matrix} \right\} \leq U$$

The intuitive meaning of such a weight constraint is that it is satisfied in a configuration, if the sum of weights of the set of *ground literals*, obtained by *grounding* the literals in the weight constraint for which both the proper and conditional parts are satisfied, is between the upper and lower bounds, inclusive. A ground literal is a literal with no variables. The set of literals obtained by grounding a literal of the form $p(\vec{x}) : q(\vec{y})$ is the set of ground literals obtained by substituting consistently the local variables in $\vec{x}$ and $\vec{y}$ with ground terms in every possible way.

Example 2.17 Consider again **Example 2.16**. The following rule set utilizing local variables is equivalent to the previous rules:

$$\begin{array}{r} slot(s_1) \leftarrow \quad slot(s_2) \leftarrow \quad slot(s_3) \leftarrow \\ card(c_1) \leftarrow \quad card(c_2) \leftarrow \quad card(c_3) \leftarrow \\ card(c_4) \leftarrow \quad card(c_5) \leftarrow \\ 0 \leq \{conn(S,C) : slot(S)\} \leq 3 \leftarrow card(C) \\ \leftarrow slot(S), 2 \leq \{conn(S,C) : card(C)\} \end{array}$$

The first eight rules again represent the different slots and cards. The ninth rule states that every card is connected to from zero to three slots. The tenth rules states that for each slot it holds that it must not have two or more connected cards. To illustrate the difference between local and global variables, consider the ninth rule. Its grounding is obtained by first substituting each global variable in the rule, in this case C, with every possible ground term as follows:

$$\begin{gathered}
0 \le \{conn(S,c_1) : slot(S)\} \le 3 \leftarrow card(c_1) \\
0 \le \{conn(S,c_2) : slot(S)\} \le 3 \leftarrow card(c_2) \\
0 \le \{conn(S,c_3) : slot(S)\} \le 3 \leftarrow card(c_3) \\
0 \le \{conn(S,c_4) : slot(S)\} \le 3 \leftarrow card(c_4) \\
0 \le \{conn(S,c_5) : slot(S)\} \le 3 \leftarrow card(c_5) \\
0 \le \{conn(S,s_1) : slot(S)\} \le 3 \leftarrow card(s_1) \\
0 \le \{conn(S,s_2) : slot(S)\} \le 3 \leftarrow card(s_2) \\
0 \le \{conn(S,s_3) : slot(S)\} \le 3 \leftarrow card(s_3)
\end{gathered}$$

The resulting set has only local variables, in this case only *S*. Note that the bodies of the last three rules cannot be satisfied, so they can never introduce anything to a configuration. Next, the local variables within weight constraints are substituted with every possible ground term, which removes the rest of the variables. Doing this for the first rule in the previous set results in the following ground rule:

$$0 \le \begin{Bmatrix} conn(c_1,c_1) : slot(c_1), conn(c_2,c_1) : slot(c_2), \\ conn(c_3,c_1) : slot(c_3), conn(c_4,c_1) : slot(c_4), \\ conn(c_5,c_1) : slot(c_5), conn(s_1,c_1) : slot(s_1), \\ conn(s_2,c_1) : slot(s_2), conn(s_3,c_1) : slot(s_3) \end{Bmatrix} \le 3 \leftarrow card(c_1)$$

To see the equivalence of the first order form of the ninth rule and its propositional form in **Example 2.16**, note that since the condition part of the literals is only satisfied for slot objects, only the literals grounded with slot objects will in effect remain within the cardinality constraint:

$$0 \le \begin{Bmatrix} conn(s_1,c_1) : slot(s_1), \\ conn(s_2,c_1) : slot(s_2), conn(s_3,c_1) : slot(s_3) \end{Bmatrix} \le 3 \leftarrow card(c_1)$$

■

2.5.5 Domain Restricted Weight Constraint Rule Language

In this section, a slightly restricted version of **FOWCRL** called domain restricted weight constraint rule language (**DRWCRL**) is presented. It is motivated by the intended use of conditional parts of literals to represent different types of objects in the configuration and efficient implementation of such a representation (I). It also allows a convenient way to define the weights of literals in weight constraints using rules.

The underlying idea of **DRWCRL** is to separate the predicates appearing in the set of rules in two categories: *domain predicates* and *general predicates*. Domain predicates are intended to represent the types, classes or sorts of objects that can appear in a configuration. General predicates are used to represent their general relations. In **Example 2.17** the predicates *slot* and *card* can be considered domain predicates defining the types of objects in a configuration, whereas the predicate *conn* is a general predicate.

Since in configuration knowledge the types of objects that can appear in a configuration are well known, it seems reasonable not to provide too much expressive power for defining domain predicates. The crucial issue here is that each object can be considered to have a known set of types, and thus the model of the domain predicates is uniquely defined for any configuration model. There is no need to express incomplete knowledge on the domain predicates. Thus, defining domain predicates using rules can be restricted so that for any set of such rules there is at most one model that is efficiently computable. This restricts the expressive power of such rules but is enough for representing types. There are several ways of doing this, of which one rather general scheme, stratified domain predicates, is presented here.

A *stratified-domain predicate* is a predicate defined by stratified normal logic program rules only. A predicate is *defined* by the subset of rules in whose head the predicate appears, and by the rules in whose heads the predicates in the bodies of the rules of the first set appear, and so on recursively. A *normal logic program rule* is a rule with only literals (negative or positive) in the body and with exactly one atom in the head of the rule. A set of normal logic program rules is *stratified* if no predicate is negatively recursively defined. A predicate is negatively recursively defined if in some rule in the rule set that defines it the same predicate appears in a negated literal. A set of such rules has a uniquely determined efficiently computable model under the semantics of weight constraint rules, as preferred for domain predicates. This holds since the semantics coincides with the stable model semantics of normal logic programs (Przymusinska and Przymusinski 1990).

The types of rules allowed in **DRWCRL** can now be defined. These are *stratified-domain restricted weight constraint rules*, for which the following holds: each variable in a rule must appear in a positive domain predicate in the body of the same rule or in the conditional part of some conditional literal in the rule. Furthermore, each condition part of a conditional literal is a stratified-domain predicate. The rules in **Example 2.17** fulfill this condition and are therefore **DRWCRL** rules.

Stratified-domain predicates can also be used to allow compact definition of the weights of literals in weight constraints. This is done by allowing a variable symbol in the place of a weight of a literal in a weight constraint, with the restriction that the variable symbol must occur in a stratified-domain predicate that occurs in the conditional part of the literal. Each literal must also have a unique weight defined in this manner.

Example 2.18 The configuration model of **Example 2.15** can be represented using **DRWCRL** as follows:

$$\begin{aligned}
prod(card_1,-20) &\leftarrow \\
prod(card_2,-20) &\leftarrow \\
prod(card_3,-40) &\leftarrow \\
prod(card_4,-40) &\leftarrow \\
prod(card_5,-80) &\leftarrow \\
prod(powersource,100) &\leftarrow \\
in(powersource) &\leftarrow \\
0 \leq \{in(C): prod(C,P) = P\} \leq 100 &\leftarrow
\end{aligned}$$

The amount that each card and power source produces and uses is represented using the stratified-domain predicate *prod* and the fact that a card or power source is in a configuration by the general predicate *in*. The grounding of the last rule, after pruning the literals for which the condition part does not hold in any configuration, is effectively as follows:

$$0 \leq \left\{\begin{array}{l}
in(card_1): prod(card_1,-20) = -20, \\
in(card_2): prod(card_2,-20) = -20, \\
in(card_3): prod(card_3,-40) = -40, \\
in(card_4): prod(card_4,-40) = -40, \\
in(card_5): prod(card_5,-80) = -80, \\
in(powersource): prod(powersource,100) = 100,
\end{array}\right\} \leq 100 \leftarrow$$

■

2.5.6 First-order Semantics

Introducing first-order rules necessitates redefining a configuration. A configuration model and a set of requirements can remain the same, i.e. two sets of rules. A configuration, on the other hand, can no longer be just any set of atoms.

A starting point for defining a configuration is provided by the notion of a *model* of a first-order language (Hodges 1983). Such a model is usually defined by defining an *interpretation* for each symbol in the language, i.e. logical connectives, object and predicate constants and variables. An interpretation represents a possible situation or state of things in the domain of interest. Therefore, the interpretation of object, predicate and variables symbols may vary, but the interpretation of logical connectives is fixed. A model of a set of sentences in the first order language is an interpretation which satisfies the sentences in the set. Thus, it is natural to define a configuration as a type of model of the set of rules in a configuration model, similarly as for propositional rules.

Defining an interpretation usually relies on the notion of a *structure*. A structure specifies a set of objects called the *domain* to which object and variable symbols may refer. It also specifies to which objects in the domain the object constant symbols refer and to which relations on the objects in the domain the predicate symbols refer. In addition to a structure, it is necessary to specify to which object in the domain a variable symbol refers to by means of an *assignment*.

In general, structures and assignments may not give a unique interpretation for all symbols in the language. Alternatively, it may be that there are elements in the domain or relations over them for which there no are symbols in the language. For configuration knowledge representation, the object constant and predicate symbols appearing in the configuration model play a role in the definition of a correct configuration. This is because it is undesirable to have objects in a configuration or relations between them that are not mentioned in the configuration model. The configuration model is supposed to represent all the necessary knowledge on the elements and their dependencies, after all. Restricting the attention to these types of models is accomplished in the following by using the notion of Herbrand models, as is usual for logic program type rules (Lloyd 1987; Przymusinska and Przymusinski 1990; Dix 1995).

Given a set of first order sentences, its *Herbrand universe* is the set of all ground terms that can be formed from the constants and function symbols in the set. For a configuration model, represented as a set of weight constraint rules, its Herbrand universe is simply the set of constant symbols appearing in the rules. A *Herbrand base* of a configuration model is the set of ground atoms, i.e. atoms without variables, constructible from the predicate symbols appearing in the configuration model by substituting each variable consistently with all members of its Herbrand universe.

Example 2.19 The Herbrand universe of the configuration model in **Example 2.17** is $\{s_1, s_2, s_3, c_1, c_2, c_3, c_4, c_5\}$. Its Herbrand base is as follows:

$$\left\{\begin{array}{l} slot(s_1), slot(s_2), \ldots, slot(c_4), slot(c_5), \\ card(s_1), card(s_2), \ldots, card(c_4), card(c_5), \\ conn(s_1, s_1), conn(s_1, s_2), \ldots, conn(s_1, c_4), conn(s_1, c_5), \\ conn(s_2, s_1), conn(s_2, s_2), \ldots, conn(s_2, c_4), conn(s_2, c_5), \\ \ldots, \\ conn(c_5, s_1), conn(c_5, s_2), \ldots, conn(c_5, c_4), conn(c_5, c_5) \end{array}\right\}$$

■

A *Herbrand interpretation* of a configuration model represented in **FOWCRL** or **DRWCRL** is an interpretation in which every object constant is interpreted as the object with the same name. This allows the identification of interpretations of a configuration model with subsets of its Herbrand base. The ground atoms in the subset of the Herbrand base are those that are true in the Herbrand interpretation, and all other ground atoms are false. A (possibly incorrect) *configuration* is then naturally defined as a Herbrand interpretation of a configuration model. A *correct configuration* is defined as a Herbrand interpretation that satisfies the rules and is grounded by the rules in the configuration model, i.e. a *stable Herbrand model*[6] of the set of rules in the configuration model.

Example 2.20 Consider again the configuration model in **Example 2.17**. The configuration, given as the subset of the Herbrand base of the configuration model,

$$\left\{\begin{array}{l} slot(s_1), slot(s_2), slot(s_3), \\ card(c_1), card(c_2), card(c_3), card(c_4), card(c_5), \\ conn(s_1, c_4), conn(s_1, c_5) \end{array}\right\}$$

is incorrect, since the tenth rule that only one card should be connected to a slot is not satisfied. The configuration

[6] Because of the established terminologies in the research traditions on configuration and logic, the word model is here used in a slightly confusing manner.

$$\left\{\begin{array}{l} slot(s_1), slot(s_2), slot(s_3), \\ card(c_1), card(c_2), card(c_3), card(c_4), card(c_5), \\ conn(s_1, c_4) \end{array}\right\}$$

is correct since it satisfies the rules in the configuration model and is grounded.
■

2.6 Formal Semantics of the Languages

The semantics of all the languages in Sections 2.3 to 2.5 formally capture the idea that a correct configuration must not contain anything that cannot be grounded in the configuration model. The formal definition of this property of configuration knowledge representation languages has been largely neglected with some. It has been identified as important in for instance the research on dynamic constraint satisfaction problems (DCSP)(Mittal and Falkenhainer 1990) and generative constraint satisfaction problems (GCSP) (Stumptner and Haselböck 1993).

The groundedness property is formalized in the semantics of the languages presented in this work using a technique similar to those developed in the research on non-monotonic reasoning and semantics of logic programs (Przymusinska and Przymusinski 1990). Only the basic outline of the technique is given here. More details can be found in (I-III).

The semantics are all based on a very similar two-part definition of a correct configuration. The first part of the definition states that the rules or constraints in the configuration model are satisfied. However, this is not enough, as a configuration with unnecessary elements can also satisfy the rules or constraints. Thus, the second part is needed. It is based on a fixpoint equation on the lattice formed by the possible configurations and their subset relation. For **CRL** and **PWCRL**, the possible configurations are all the subsets of the atoms in the rules representing a configuration model. For **FOWCRL** and **DRWCRL**, the possible configurations are all the subsets of the Herbrand Base of the configuration model. Finally, for DCSP, the possible configurations are the subsets of all the possible assignments of values to variables.

The fixpoint condition in the definition is a way of ensuring that the rules or activity constraints and initial variables ground all elements in a correct configuration. To capture this, a *reduction* of the rules or activity constraints and initial variables with respect to a configuration is defined. The intuition behind the reduction is that it produces for each element in a configuration a set of simpler rules or activity constraints that can ground the element. The whole set of rules thus obtained is called a *reduct*. If some element is not in a configuration, then there is no need for it to be grounded and consequently no corresponding simpler rule or activity constraints are included in the reduct.

Then, an operator $\mathrm{T}(.)$ on the lattice formed by the set of all possible configurations and the subset relation on these is defined. Intuitively, the operator captures how the simpler rules or activity constraints introduce new elements to a configuration when their bodies or left hand side constraints, are grounded and satisfied.

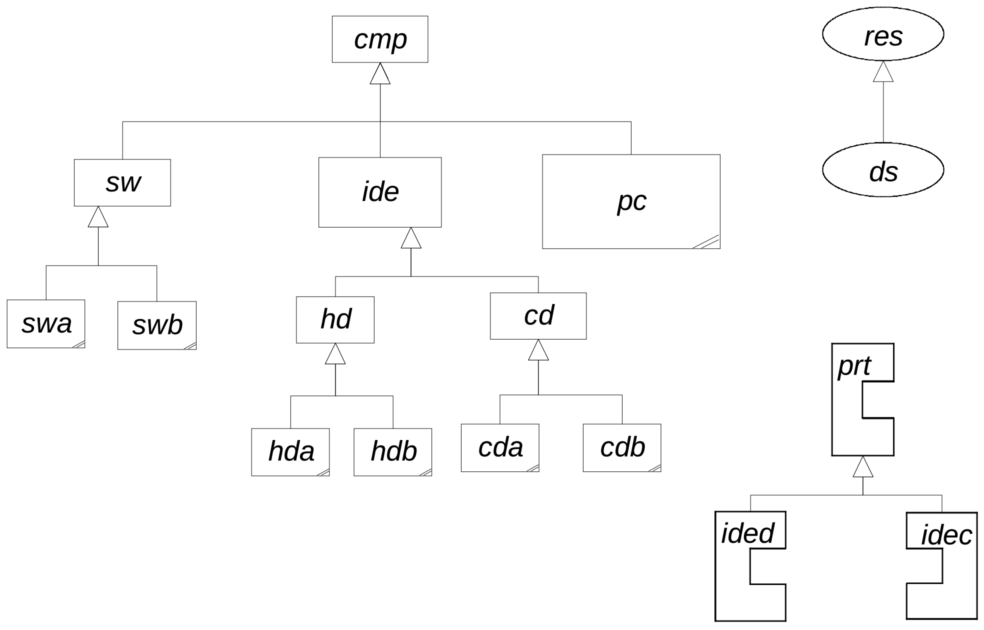
A correct configuration contains all and only the elements that are grounded by the rules. Since elements may be justified recursively, a correct configuration must be a fixpoint of the above-defined operator for the reduct of the configuration model with respect to the configuration. A *fixpoint configuration* q of an operator $\mathrm{T}(.)$ is such that $\mathrm{T}(q)=q$. This ensures that every element with a (possibly recursive) ground is included in the configuration, as the operator will only then produce the same configuration as its input.

For the semantics of the languages in this work, the operators $\mathrm{T}(.)$ are monotonic, i.e., for any two configurations q_1 and q_2 it holds that $q_1 \subseteq q_2$ implies $\mathrm{T}(q_1) \subseteq \mathrm{T}(q_2)$ (Lloyd 1987). A monotonic operator has a unique *least fixpoint*. As a correct configuration must not contain ungrounded elements, it is defined as such a least fixpoint. This ensures that all elements in a correct configuration solution are grounded by the simplified rules or activity constraints, thus formalizing the groundedness.

2.7 Formalizing the Simplified Configuration Ontology

In this section the formal treatment of configuration knowledge representation is continued by representing the PC configuration model presented in Section 2.2 using **DRWCRL** introduced in Section 2.5 (IV). As the main interest in this work is on the configuration task, in the following it is assumed that the configuration model is *correct* with respect to the ontology in the sense that the types, their property definitions and constraints in it are allowed by the ontology.

In the representation, configuration model knowledge, i.e. a configuration model, is represented as a set of **DRWCRL** rules. Configuration solution knowledge, i.e. a configuration, is then defined as a stable Herbrand model of the set of rules in the configuration model. Requirements knowledge, i.e. the requirements on a configuration, is defined as another set of **DRWCRL** rules that a configuration must satisfy. A distinction is made between the rules giving *ontological definitions* and the rules representing a configuration model. The former are not changed when defining a new configuration model. The latter appear only in the formalization of the example. Moreover, domain predicates giving the taxonomy of the configuration model, i.e. the types of objects that may appear in a configuration, are typeset normally whereas other predicates defining the configuration are typeset in ***boldface***.



(a)

$$
\begin{array}{lclcl}
cmp(C) \leftarrow pc(C) & ; & ide(C) \leftarrow hd(C) & ; & hd(C) \leftarrow hda(C) \\
cmp(C) \leftarrow ide(C) & ; & ide(C) \leftarrow cd(C) & ; & hd(C) \leftarrow hdb(C) \\
cmp(C) \leftarrow sw(C) & ; & sw(C) \leftarrow swa(C) & ; & cd(C) \leftarrow cda(C) \\
 & & sw(C) \leftarrow swb(C) & ; & cd(C) \leftarrow cdb(C)
\end{array}
$$

$$
\begin{array}{l}
res(R) \leftarrow ds(R) \\
prt(P) \leftarrow idec(P) \\
prt(P) \leftarrow ided(P)
\end{array}
$$

(b)

Figure 2.4 (a) Taxonomy of the PC configuration model. (b) Rule representation of the taxonomy.

2.7.1 Types, Individuals and Taxonomy

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 the definition of 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 means that any rules on the individuals of the supertype are also applicable to the individuals of the subtypes, which effects the monotonic inheritance of the property definitions.

Example 2.21 The taxonomy of the PC configuration model is represented using rules in **Figure 2.4**.

■

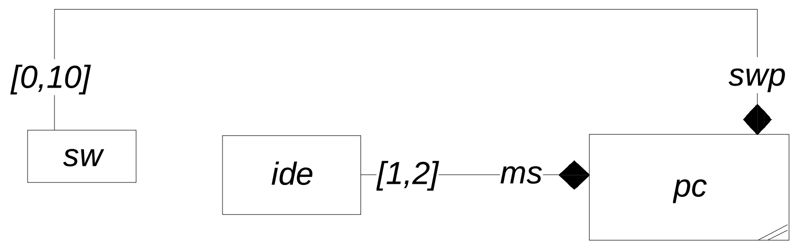
2.7.2 Compositional Structure

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 is collected 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 2.22 The root component type pc, the part names, possible part types and the part definitions of pc are represented using rules in **Figure 2.5**.

■



(a)

$$root(C) \leftarrow pc(C)$$

$$\begin{aligned}
pan(ms) &\leftarrow \\
ppa(C_1, C_2, ms) &\leftarrow pc(C_1), ide(C_2) \\
1\{\boldsymbol{pa}(C_1, C_2, ms) : ppa(C_1, C_2, ms)\}2 &\leftarrow \boldsymbol{in}(C_1), pc(C_1)
\end{aligned}$$

$$\begin{aligned}
pan(swp) &\leftarrow \\
ppa(C_1, C_2, swp) &\leftarrow pc(C_1), sw(C_2) \\
0\{\boldsymbol{pa}(C_1, C_2, swp) : ppa(C_1, C_2, swp)\}10 &\leftarrow \boldsymbol{in}(C_1), pc(C_1)
\end{aligned}$$

(b)

Figure 2.5 (a) Structure of the PC configuration model. (b) Rule representation of the structure.

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 represented as follows:

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

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

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

2.7.3 Resources

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 individual of the producing or using component type, a component individual of the produced or used resource type, and the magnitude. Use is represented as negative magnitude.

Example 2.23 The production and use definitions in the PC configuration model are represented using rules in **Figure 2.6**.

■

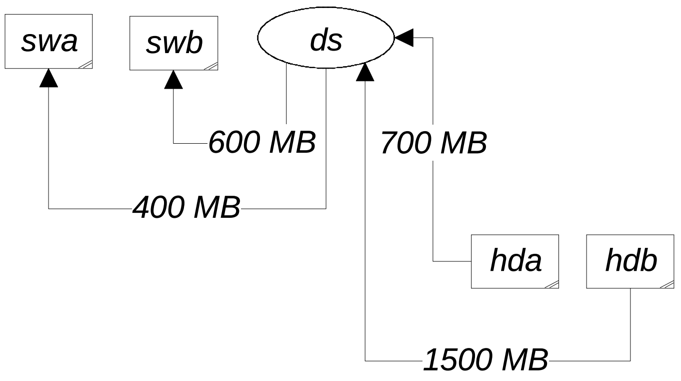
The production and use of a resource type by the component individuals is represented as weights of the predicate $\boldsymbol{in}(\ )$. The ontological definition that the resource use must be satisfied by the production is expressed with a weight constraint rule stating that the sum of the produced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

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

2.7.4 Ports and Connections

Port types are represented as domain predicates and port individuals as uniquely named object constants. The compatibility of port types is represented as the binary domain predicate $cmb(\ )$ on port individuals of compatible port types and as 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:

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



(a)

$$prd(C, ds, -400) \leftarrow swa(C) \quad ; \quad prd(C, ds, 700) \leftarrow hda(C)$$
$$prd(C, ds, -600) \leftarrow swb(C) \quad ; \quad prd(C, ds, 1500) \leftarrow hdb(C)$$

(b)

Figure 2.6 (a) Resource production and use definitions in the PC configuration model. (b) Rule representation of the resource production and use definitions.

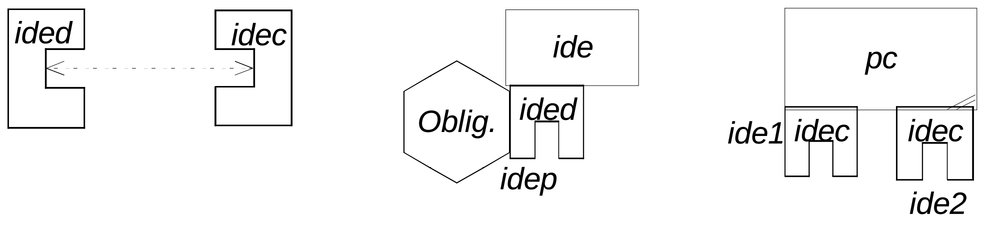
A port definition of a component type is represented as a rule very similarly to a part definition, but with the tertiary predicate $po(\ )$ signifying that a component individual has a port individual with a given port name. The $pon(\ )$ domain predicate captures the port names.

Example 2.24 The compatibility definitions of port types and the port definitions are represented using rules in **Figure 2.7**.

■

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:

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



(a)

$$cmb(P_1, P_2) \leftarrow idec(P_1), ided(P_2)$$
$$cmb(P_1, P_2) \leftarrow ided(P_1), idec(P_2)$$

$$pon(ide1) \leftarrow \quad ; \quad 1\{\boldsymbol{po}(C, P, ide1) : idec(P)\}1 \leftarrow \boldsymbol{in}(C), pc(C)$$

$$pon(ide2) \leftarrow \quad ; \quad 1\{\boldsymbol{po}(C, P, ide2) : idec(P)\}1 \leftarrow \boldsymbol{in}(C), pc(C)$$

$$pon(idep) \leftarrow \quad ; \quad 1\{\boldsymbol{po}(C, P, idep) : ided(P)\}1 \leftarrow \boldsymbol{in}(C), ide(C)$$
$$\leftarrow ide(C), ided(P_1), \boldsymbol{po}(C, P_1, idep), \{\boldsymbol{cn}(P_1, P_2) : prt(P_2)\}0$$

(b)

Figure 2.7 (a) The compatibility definitions of port types and the port definitions in the PC configuration model. (b) Rule representation of the compatibility definitions and port definitions.

2.7.5 Constraints

Rules without heads are used to represent constraints.

Example 2.25 The constraint that a hard disk of type hd must be part of pc is represented using the following rule:

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

■

2.8 Analysis of the Formalisms

In this section, the computational complexity of configuration related tasks for the formalisms and the simplified configuration ontology presented above is briefly analyzed. In addition, some results on the expressivity of the formalisms in relation to each other are given. Mostly only the results are presented here. More details on their proofs and definitions can be found in (I-IV).

2.8.1 Computational Complexity

The analysis is based on the standard techniques and theory on computational complexity. Only a brief overview of this topic is given here; more information can be found in (Papadimitriou 1994). Recall that the usual complexity classes according to which the computational problems are classified reflect how hard the problem is in the worst case, for example for the most difficult configuration model and requirements. A problem which is *complete* for a complexity class is among the computationally most difficult problems in that class. A problem that is *hard* with respect to a complexity class is at least as hard as the problems in that class, but may be harder, i.e. in a higher complexity class.

Problems in the complexity class *polynomial time*, **P**, are considered to be *tractable*, i.e., efficiently computable even in the worst case. As the name implies, the time taken by a computation for a problem in the worst case in this class is bounded by a polynomial function in the size of the problem. The class **P** is a subclass of the class *non-deterministic polynomial time*, **NP**, and believed to be a strict subclass[7]. The potential problems that are in **NP** or **NP**-hard but not in **P** are believed to be *intractable*, i.e. that the time taken by a computation in the worst case is bounded by an exponential function in the size of the problem, and thus infeasible for efficient computation. Classes **P** and **NP** are both contained in the class exponential time, **EXP**. The problems in this class and not in a lower class are provably intractable, i.e. in the worst case their computation takes an exponential amount of time in the size of the problem. The class of *decidable* problems consists of all the problems that can be solved using a computer, whereas a computer cannot solve *undecidable* problems.

The two generic problems related to configuration whose complexity will be analyzed for each of the formalisms are defined in the following.

[7] It is not known if $\mathbf{P} = \mathbf{NP}$, although the general belief is that this is not the case.

Table 2.2 Complexity of configuration related tasks

L_1	L_2	CHECK	CONF(D)
CRL	**CRL**	In **P** (III)	**NP**-complete (III)
PWCRL	**PWCRL**	In **P** (I)	**NP**-complete (I)
FOWCRL	**FOWCRL**	?	**EXP**-hard, decidable
DRWCRL	**DRWCRL**	?	**EXP**-hard, decidable
$\mathbf{DRWCRL_{Ont}}$	**DRWCRL**	In **P**	**NP**-complete
CSP	CSP	In **P** (Mackworth 1977)	**NP**-complete (Mackworth 1977)
DCSP	DCSP	In **P** (II)	**NP**-complete (II)
EDCSP	EDCSP	In **P** (II)	**NP**-complete (II)

Definition 2.1 (Problem Definitions)

CONFIGURATION CHECKING (CHECK): Given a finite configuration model CM represented using formalism L_1 and a finite configuration C with respect to L_1, is C correct with respect to CM ?

CONFIGURATION(D) (CONF(D)): Given a finite configuration model CM represented using formalism L_1 and a finite set of requirements R on the configuration, represented using formalism L_2, is there a configuration C with respect to L_1, such that C is correct with respect to CM and satisfies R?

The latter problem is the decision (denoted by (D)) version of the configuration problem. Recall that a problem is at least as hard as its decision version. Thus, the decision version can be used as an abstraction that allows a simpler complexity analysis while still characterizing the hardness of the problem. Therefore, the decision version is used in the following.

The definitions are parameterized with respect to the formalisms in which the configuration model and the requirements are represented, and which define the form of a configuration. For simplicity, it is mostly assumed that the requirements are represented using the same formalism[8] as the configuration

[8] The word formalism is used here to mean the generic formalism, without committing to a specific set of symbols other than the fixed ones in the formalism. Therefore, the requirements can contain terms that do not appear in the configuration model. To be precise, one should speak of families of formalisms, parameterized by the terms appearing in a configuration model or requirements, instead of a formalism.

model. The complexity is measured with respect to the combined size of the configuration model and configuration for CHECK and with respect to the combined size of the configuration model and requirements for CONF(D), except for **DRWCRL$_{\mathbf{Ont}}$** as discussed below.

In **Table 2.2** the complexities of the tasks for the different formalisms are given. The formalism name **DRWCRL$_{\mathbf{Ont}}$** refers to the case where the configuration model is represented using **DRWCRL** based on the formalization of the simplified ontology presented in Section 2.7 and given two additional assumptions discussed below. The hardness and decidability results for the CONF(D) problem for **FOWCRL**, **DRWCRL** and **DRWCRL$_{\mathbf{Ont}}$** are shown below. Note that the CHECK problem has not been analyzed for **FOWCRL** and **DRWCRL** and that the CONF(D) problem has not been precisely classified with respect to them.

Theorem 2.1 (Complexity of FOWCRL) CONF(D) with $L_1 = L_2 =$ **FOWCRL** is decidable and **EXP**-hard.

Proof: Decidability can be shown as follows: By the definition of **FOWCRL**, a configuration with respect to a configuration model CM represented in **FOWCRL** is a subset of the Herbrand base HB_{CM} of CM. Since the Herbrand universe HU_{CM} of CM consists of the object constants in CM, it is finite. Therefore HB_{CM}, constructed by instantiating each of the finite set of predicate symbols in CM by every member of HU_{CM} is finite as well. A naïve algorithm for CONF(D) is obtained as follows: given CM and a set of requirements R, construct HU_{CM} and HB_{CM}. Ground CM and R by consistently substituting every variable in CM and R by the objects in HU_{CM} in every possible way. Then, for each subset of HB_{CM}, check if it is correct, i.e. a stable model of the grounded CM that satisfies the grounded CM, and satisfies the grounded R. If such a model is found, then the answer is "yes", otherwise "no". The algorithm clearly terminates in finite time with the correct answer, since HB_{CM} is finite.

The **EXP**-hardness can be shown by using the problem of whether a ground atom a is true in every Herbrand model of a first-order positive Horn clause program H without function symbols. For this problem the expression complexity, i.e. the complexity measured in the size of the rules in H that are not ground atomic facts and do not contain constant symbols, is **EXP**-complete with respect to the van Emden-Kowalski semantics (e.g. (Schlipf

1995)). Since a positive first-order Horn clause program has exactly one Herbrand model with respect to the van Emden-Kowalski semantics, this problem is equivalent to the problem of whether the ground atom a is true in some Herbrand model of H. This problem can in turn be reduced to CONF(D), since for Horn clause programs the stable model semantics of **FOWCRL** coincides with the van Emden-Kowalski semantics. The reduction is simply as follows: let $CM = H$ and $R = \{a \leftarrow\}$. Now, there is a model of H in which a is true if and only if there is a correct configuration C, i.e. a stable model of CM satisfying CM, such that C satisfies R. Thus, CONF(D) is **EXP**-hard for **FOWCRL**.

■

Since **DRWCRL** is a special case of **FOWCRL** in which Horn clause programs can be represented using exactly one domain predicate which holds for every object constant in the Horn clause program and is used as the domain predicate for each variable, the following corollary is obtained:

Corollary 2.1 (Complexity of DRWCRL) CONF(D) with $L_1 = L_2 =$ **DRWCRL** is decidable and **EXP**-hard.

■

The relatively high complexity of **FOWCRL** and **DRWCRL** seems to indicate that these formalisms capture harder configuration tasks than do CSP or DCSP. This is due to allowing rules with variables that encode very compactly large (exponential in the number of variables in a rule) sets of ground rules constructed out of predicate symbols. The variables with their domains and constraints of CSP or DCSP correspond to such ground predicates, and thus rules with variables can encode them very compactly. Therefore, it is no longer evident that CHECK is computationally efficient, i.e. in **P**.

However, for the formalization of the simplified configuration ontology in Section 2.7 and given certain additional assumptions, CHECK remains efficient. The set of **DRWCRL** rules resulting from the formalization of the ontology and a configuration model, while respecting the additional assumptions, is called **DRWCRL$_{\text{Ont}}$**. Next the intuition behind this result is presented and results on the computational complexity of the configuration related tasks based on the simplified ontology and the additional restrictions is analyzed.

To facilitate the analysis, the following basic assumptions are made. A configuration model $\underline{CM}$ represented according to the ontology is translated to a set of **DRWCRL** rules CM as in Section 2.7, 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 as follows: add a fact $t(j) \leftarrow$ for each individual j whose concrete type is t. The set of rules $CM \cup S$ is all that is needed to represent the product, and subsequently configure it. The requirements are represented using a set of **DRWCRL** rules R.

The reason for introducing the set S is as follows. It can be thought of as representing 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, and hence the CHECK task very hard. The first assumption is thus as follows:

Assumption 1 There is a pre-computed set S representing the individuals of concrete types that may be in a correct configuration with respect to $\underline{CM}$ and this set is added to CM .

There is a second source of complexity for the configuration task arising from the power of the rules with variables. To address this, the following assumption is made:

Assumption 2 The number of variables in the rule representation of each constraint in $\underline{CM}$ and each rule in R is bounded by some constant c_v.

This assumption is based on the observation that even checking whether a constraint rule or requirement rule of arbitrary length is satisfied by a configuration may be computationally very hard. In most CSP approaches, for example, only binary constraints, i.e. relations with two variables, are allowed. Since the ontological definitions have at most five variables, this assumption implies that there is a bound $c = \max(5, c_v)$ on the number of variables in any rule of any CM and R.

The effect of the two assumptions is to bring the complexity of the configuration task to a lower class than for unrestricted **DRWCRL**.

Theorem 2.2 (IV) CONF (D) for **DRWCRL$_{\mathbf{Ont}}$** is **NP**-complete with respect to the size of $CM \cup S \cup R$.

■

The checking task can similarly be shown efficiently computable (IV).

Corollary 2.2. CHECK for **DRWCRL$_{\mathbf{Ont}}$** is in **P** in the size of $CM \cup S \cup C$.

■

2.8.2 Expressiveness

Most of the formalisms addressed above have the same computational complexity for the configuration task. This indicates that a problem in one formalism can be translated to another formalism efficiently, i.e. in polynomial time with the resulting problem being of polynomial size with respect to the original, and then solved using that formalism. A natural question is to ask whether there are any differences between the formalisms, or could one just use any of them? To answer this question, a more fine-grained analysis of the properties of the formalisms than that provided by complexity analysis is needed.

One concept that aids in this is *modularity*. This property is important when the knowledge represented using the formalism needs to be changed. Intuitively, a modular representation of knowledge is such that a small change to the knowledge results in a small change to the representation. The essential concept here is the notion of a *modular representation* of a problem originally given in one formalism in another formalism. Such a representation allows two disjoint pieces of the original problem to be represented so that if one of them is changed, then only the representation of that piece needs to be changed, while the representation of the other stays the same. In other words, the two pieces can be represented independently of each other.

A modular representation must of course be faithful, meaning that the solutions to the representation agree with the solutions to the original problem. Note that even though two decision problems are in the same complexity class, such a faithful polynomial translation may not exist, since it is enough that there is translation that preserves the answer to the decision problem. It is also required to be polynomial, meaning that the representation can be computed using a systematic *translation* in polynomial time and result in a polynomial problem size with respect to the original problem size. Note that the latter property implies that a formalism for which a problem of interest is hard for a complexity class cannot be modularly represented by a formalism for which the problem is in a provably lower complexity class.

One can now compare two formalisms by studying whether one of them can be modularly represented using the other, i.e. if there is a systematic, modular, faithful, and polynomial translation from one of them to the other. If formalism L_1 can be modularly represented by L_2 but not the other way around, then L_2 allows a more easily changed representation of knowledge and can be argued to be a more expressible and therefore preferable formalism.

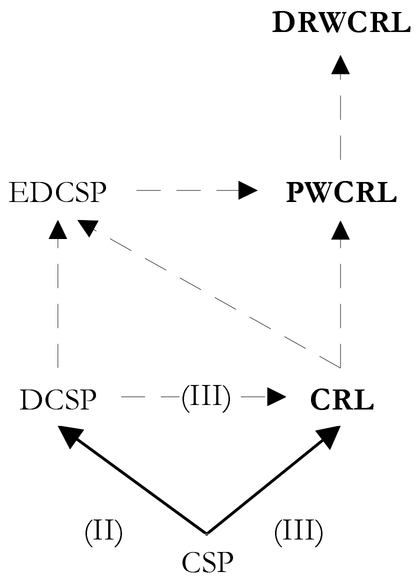


Figure 2.8 Expressiveness of the formalisms

In **Figure 2.8** some of the formalisms are compared from the modularity point of view. A dashed arrow from formalism L_1 to formalism L_2 states that L_2 can modularly represent L_1, but the other direction has not been analyzed. In addition, some of the formalisms have not been compared at all and thus some arrows may be lacking. Therefore, the figure does not give a complete classification of the formalisms. A continuous arrow indicates that the property has been analyzed in both directions. If there is a continuous arrow in one direction only, the modularity is strict, i.e. L_1 cannot modularly represent L_2. This means that L_2 is strictly more expressive than L_1 with respect to modularity. For example, CSP can be modularly represented by DCSP, but DCSP cannot be modularly represented by CSP. DCSP is thus more expressive. Similarly, **CRL** can be modularly represented by **PWCRL**. However, the other direction has not been analyzed.

Attached to the arrow is a reference to where the property is shown. The other results are shown below. Note that the definition of what precisely constitutes a modular representation depends on the formalisms compared. In showing these results, very weak assumptions were used. For modularity, it is required that each rule, variable with its domain, initial variable, compatibility constraint and activity constraint in a problem can be represented independently.

To show the results, it is first noted that if a formalism L_1 is a generalization of L_2, i.e. that the definitions of the semantics of L_1 coincide with the definition of the semantics of L_2 for that subclass of L_1 which is the same as L_2, then L_1 can obviously represent modularly L_2. This observation provides the following results.

Theorem 2.3 (Modularity Results by Generalizations) DCSP can be modularly represented by EDCSP. **CRL** can be modularly represented by **PWCRL**, which in turn can be modularly represented by **DRWCRL**.

Proof: Any DCSP D can be transformed into an EDCSP E by leaving the sets of variables and their domains, initial variables and compatibility constraints as they are. For each activity constraint

$$c_i \overset{ACT}{\rightarrow} v_j$$

in D add to E the activity constraint

$$c_i \overset{ACT}{\rightarrow} 1\{v_j\}1$$

Any set of **CRL** rules r can be modularly represented by a set of **PWCRL** rules w using the translation outlined in Section 2.5.3. In addition, any set of **PWCRL** rules can be modularly represented by a set of **DRWCRL** rules, since a propositional atom can be seen as an atom with a zero-ary predicate.

■

Another way of obtaining modularity results is to consider the setting where L_1 is modularly representable by L_2 and L_2 is modularly representable by L_3. This gives a modularity result for L_1 with respect to L_3.

Theorem 2.4 (Transitivity of Modularity Results) If L_1 is modularly representable by L_2 and L_2 is modularly representable by L_3, then L_1 is modularly representable by L_3.

Proof: Assume that there are modular translations t_1 from L_1 to L_2 and t_2 from L_2 to L_3. For the result to hold there must be a translation t_3 from L_1 to L_3 such that the translation is faithful, modular and polynomial. Since t_1 and

t_2 are faithful, their composition $t_2(t_1(.))$ is obviously also faithful. It can also be computed in polynomial time by first applying t_1 and then t_2 to the result. The composition can be shown to be modular by noting that any two pieces of a problem represented in L_1 are translated to two independent pieces in L_2, which in turn are both also translated to two independent pieces in L_3. Now, a change to either piece in L_1 clearly only affects its representation in L_3.

■

This result means that the diagram in **Figure 2.8** implies that CSP can for instance be modularly represented as EDCSP and DCSP as **PWCRL**. The final results in this section show that **CRL** can be modularly represented using EDCSP and EDCSP using **PWCRL**.

Theorem 2.5 (CRL vs. EDCSP) CRL can be modularly represented by EDCSP.

Proof: Any set of **CRL** rules R can be transformed into an equivalent EDCSP E by the following translation. For each proposition p appearing in R, introduce a variable with the same name and with a unary domain in E. Let the sets of initial variables and compatibility constraints in E be empty. For each rule

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

in R, add the activity constraint

$$b_1^c, \ldots, b_m^c, \text{not}(c_1^c), \ldots, \text{not}(c_n^c) \rightarrow 1\{a_1 \mid \cdots \mid a_l\}l$$

and for each rule

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

the activity constraint

$$b_1^c, \ldots, b_m^c, \text{not}(c_1^c), \ldots, \text{not}(c_n^c) \rightarrow 1\{a_1 \mid \cdots \mid a_l\}1$$

in E, where each of $b_1^c, \ldots, b_m^c$ and $c_1^c, \ldots, c_n^c$ is a constraint that the variable corresponding to the proposition has the (unique) value in its domain.

Now the solutions to E correspond to the stable models of R by the definitions of EDCSP and **CRL** (II,III). The active variables in a solution to E provide the atoms satisfied in a model of R. The translation is thus faithful. It can also be carried out in polynomial time. Finally, it is also modular, since each rule is translated independently of other rules.

■

Theorem 2.6 (EDCSP vs. PWCRL) EDCSP can be modularly represented by **PWCRL**.

Proof: Given an EDCSP E with the set of variables V with their domains, the set of initial variables V_I, the set of compatibility constraints C_C and the set of activity constraints C_A, construct the following set R of **PWCRL** rules:

(1) For each variable $v \in V$ with domain D consisting of the values d_i, $1 \le i \le n$, add the following rule:

$$1\{v = d_1, \ldots, v = d_n,\}1 \leftarrow act(v)$$

(2) For each initial variable $v \in V_I$, add the following fact:

$$act(v) \leftarrow$$

(3) For each compatibility constraint $c \in C_C$ on variables $v_1, \ldots, v_m$ with allowed value combinations $\{(d_{1,1}, \ldots, d_{1,m}), \ldots, (d_{n,1}, \ldots, d_{n,m})\}$, add the following rules:

$$\begin{gathered} sat(c) \leftarrow v_1 = d_{1,1}, \ldots, v_m = d_{1,m} \\ \vdots \\ sat(c) \leftarrow v_1 = d_{n,1}, \ldots, v_m = d_{n,m} \\ \leftarrow act(v_1), \ldots, act(v_m), \text{not}\, sat(c) \end{gathered}$$

(4) For each activity constraint $a \in C_A$ of the form $c_1, \ldots, c_j, \text{not}(c_{j+1}), \ldots, \text{not}(c_k) \xrightarrow{ACT} m\{v_1 \mid \cdots \mid v_l\}n$ with compatibility constraints $c_1, \ldots c_k$, each on variables $v_{1,i}, \ldots, v_{p,i}, 1 \le i \le k$, with allowed value combinations $\{(d_{1,1,i}, \ldots, d_{1,p,i}), \ldots, (d_{r,1,i}, \ldots, d_{r,p,i})\}$, add the following rules:

$$sat(c_i) \leftarrow v_{1,i} = d_{1,1,i}, \ldots, v_{p,i} = d_{1,p,i}, act(v_{1,i}), \ldots, act(v_{p,i})$$
$$\vdots$$
$$sat(c_i) \leftarrow v_{1,i} = d_{r,1,i}, \ldots, v_{p,i} = d_{r,p,i}, act(v_{1,i}), \ldots, act(v_{p,i})$$
$$m\{act(v_1), \ldots, act(v_l)\}n \leftarrow sat(c_1), \ldots, sat(c_j), \text{not}\, sat(c_{j+1}), \ldots, \text{not}\, sat(c_k)$$

Now the stable models of R correspond to the solutions of E by the definitions of **PWCRL** and EDCSP. The atoms of form $act(v)$ in the stable models encode the active variables in a solution to E and those of form $v = d_1$ the value assignments for the variables. The translation is thus faithful. It can also be carried out in polynomial (in fact, linear) time. Finally, it is also modular, since each variable with its domain, initial variable, compatibility constraint and activity constraint is translated independently.

■

2.9 Empirical Evidence for the Approach

In this section, some empirical evidence that the approach, the ontology and the formalisms in this work are relevant for practical configuration problems is provided. First, evidence for the usefulness of the concepts in the configuration ontology is provided by modeling case products. Then, a prototype implementation of the formalisms is described and test results for solving small configuration problems represented in the formalisms are given.

2.9.1 Modeling Case Products

The phenomena represented by the concepts in the generalized ontology are found in three simplifications of case products: a hospital monitor, a rock drilling machine and a PC. Thus, the concepts are useful for modeling configuration knowledge

The configuration model of a hospital monitor (V) based on the generalized ontology is a simplification loosely based on a more detailed analysis on a real configurable product for clinical measurements (Soininen 1996). The analysis was carried out based on a manual used in a manufacturing company for documenting the configuration knowledge. In the model, the main concepts related to the taxonomy, compositional structure, connections, resource interactions and functions were identified. In addition, several constraints were identified, in particular between the function types and component types im-

Table 2.3 Performance results for some configuration problems.

Problem	**First configuration**	**All configurations**	G^{first}	G^{all}
CAR	0.05 s	0.06 s	5	197
CARx2	0.05 s	3.2 s	6	44455
Monitor	0.06 s	0.31 s	9	1319
Mixer	0.04 s	0.05 s	4	87

plementing them. Most of the phenomena in the configuration model can in fact be modeled based on the simplified ontology.

The configuration model of a rock drilling machine (VI) is also based on the generalized ontology, although only the use of attributes in that model is beyond the simplified ontology. The configuration model represents a simplification of a real configurable product in which the main variation of the product is incorporated and details omitted. The model is based on interviews and an analysis of documentation on the product. Again, the main concepts related to the taxonomy, compositional structure, connections, resource interactions and functions were identified. Due to the general viewpoint of engineers on mechanical products reflected in the documentation of the product, the compositional structure played a major role in the model. Resource and connection oriented phenomena as well as to some extent functions and contexts for resource interactions were identifiable, however. Rather few constraints were needed for the model at this level of abstraction as the other concepts described the product adequately.

The configuration model of a PC (IV) based on the simplified ontology represents a very simple part of a fictional configurable product. It is easy to identify phenomena represented by the taxonomy, compositional structure, resources and connections in the PC domain.

2.9.2 Test Results for the Formalisms

In (I)-(III), prototype implementations of the configuration task for **CRL**, **PWCRL** (with only integer weights of literals and no strict inequality operators for weight constraints), **DRWCRL** (with integer weights, no strict inequality, and simpler domain predicates) and DCSP are described. The implementations are all based on defining a faithful mapping from each of these formalisms to a set of ground *constraint rules* (Simons 2000), an extension of normal logic programs with the stable model semantics. Note that the mapping from EDCSP to **PWCRL** outlined in the proof of **Theorem 2.1** lays a basis for similarly implementing EDCSP. For **CRL**, **PWCRL**, DCSP, and EDCSP the mappings produce a set of rules linear in the size of the original problem. For **DRWCRL**, the mapping in the worst case produces an exponential number of ground rules due to allowing variables in **DRWCRL**. However, by using the information provided by domain predicates in the mapping, the set of ground rules provided by the mapping often is considerably smaller than the entire grounding of the problem, while still having the same configurations, i.e. stable models (I).

Using the mappings, a configuration model represented in any of the formalisms can be translated to constraint rules. Then, the configuration task is carried out using an efficient implementation of such rules called the *Smodels* system (Simons 2000). Smodels finds the stable models of basic constraint rules using a Davis-Putnam-like backtracking search through a binary search tree where nodes are propositional atoms. The search space is very efficiently pruned by using rule propagation. In addition, the system uses a powerful dynamic application-independent search heuristic.

In **Table 2.3**, performance results are provided for the solving of several small configuration problems using this implementation approach (II). The problems, a car configuration problem (CAR), an enlarged car configuration problem (CARx2), a simplified form of the hospital monitor problem (Monitor), and a simplified form of a mixer problem (Mixer), were originally defined as DCSPs. They were manually translated to a set of **CRL** rules and then to constraint rules. Smodels was then used to find the first correct configuration and all correct configurations for this set of rules. The execution times include the time for reading the **CRL** input from a file, translating it and parsing it. As an additional characterization, the number of non-deterministic guesses G for finding the first solution and all solutions are provided. In the implementation, a non-deterministic guess chooses a propositional atom corresponding to a particular value assignment or activity of a DCSP variable. More details on the experiment set up can be found in (II). The performance of the implementation for these problems is good enough to use it as a part of a configurator in an interactive setting where the user inputs some requirements and the configurator supplies suitable configurations. This can be seen by noting that the time to provide all solutions provides a rough upper bound estimate for computing a solution to any set of requirements or finding out that there is no solution for a given set of requirements. This is because adding a set of requirements does not increase the search space and, in fact, the requirements can be used to prune the search space.

Table 2.4 Characteristics of the problems

Problem	$\lvert V\rvert$	$\lvert C_C\rvert$ / **max. arity**	$\lvert C_A\rvert$ / **max. arity**	$\max(\lvert D_i\rvert)$	$\lvert solutions\rvert$	**Search space**
CAR	8	7/3	8/1	3	198	1296
CARx2	8	7/3	8/1	6	44456	331776
Monitor	24	9/3	19/3	4	1320	196608
Mixer	8	4/2	6/1	4	88	1152

In Table 2.4 these problems are characterized by the number of variables, compatibility constraints and their maximum arity, the number of activity constraints and the maximum arity of their left hand side constraints. In addition, the maximum domain size, number of correct solutions, and size of the initial search space calculated by multiplying the domain sizes of the variables are given.

3 Discussion and Comparison with Previous Work

In this section, the approach taken in this work is discussed and compared with several formal and conceptually well-founded approaches to configuration knowledge representation and reasoning. This type of analysis has mostly been lacking in configuration research, although it is important for furthering the maturity of the field.

First, the different approaches are discussed in general and some general conclusions given. The generalized ontology is briefly compared with previous approaches and found to cover, unify and extend the major conceptualizations of configuration knowledge as structure, resource interactions, connections and functions. General observations on the importance and effect of the groundedness principle explored in this work are made. This principle differentiates the languages developed in this work from most of the other approaches by allowing a more compact and modular representation and making a clear distinction between the roles of the configuration model and requirements.

Three approaches to developing a conceptualization and formalizing it are identified and discussed. The first specifies a fixed ontology and formalizes it directly. The second approach is based on a neutral general-purpose representation language that is used to define an ontology. The third approach does not specify a detailed ontology but only very general concepts. It provides a formal language meant for configuration knowledge representation. The approach taken in this work is closest to the second approach. The difference to that and particularly the effect of the groundedness on the formalization is discussed. In the final section on general discussion and conclusions, three classes of complexity results, **NP**-complete, decidable and undecidable, into which the different approaches can be divided are identified and discussed.

Then, the formalization of the simplified ontology in this work is discussed and contrasted with several models covering either the ontology or formalization of configuration knowledge or both. The scarcity of models combining both an ontology and a formalization emphasizes the importance of this type of research. The discussion covers

- the models of Najman and Stein (Najman and Stein 1992),
- the description logic-based approaches of the PROSE system (McGuinness and Wright 1998), Constructive Problem Solving (Buchheit

et al. 1995), a formalization of the PLAKON model (Schröder et al. 1996), and the model of Owsnicki-Klewe (Owsnicki-Klewe 1988),

- the Configuration Design Ontologies (Gruber et al. 1996), and
- a UML-based conceptualization (Felfernig et al. 1999) formalized within the framework of consistency-based configuration (Friedrich and Stumptner 1999).

For each of these models its ontological foundation, the formal language used, the formalization, the computational complexity of the configuration tasks based on the formalization, and the efficiency and practical relevance of an implementation based on the approach are discussed.

However, comparison with empirical results on the efficiency and practical relevance of the implementations of these models is only briefly dealt with. This is due to the scarcity of comprehensive reports on such empirical evaluation. Mostly it is only reported that one or two products were modeled and that the model and implementation seem to work efficiently enough and support the practical needs. However, results on large-scale experiments, details on the implementations and algorithms, test set ups, detailed product models or thorough analyses of the usability of the systems are not provided. In addition, the reports on commercially applied systems do not usually provide detailed accounts on the functionality, implementation or practical relevance of the systems, probably because that type of information is confidential.

3.1 General Discussion and Conclusions

3.1.1 Conceptualizations of Configuration Knowledge

The generalized ontology presented in Section 2.1 covers the conceptualizations of the connection-, resource-, structure- and function-oriented approaches. It is the most generic ontology in this sense and extends the previous conceptualizations in several ways. It does not have a minimal number of concepts in the formal sense for representing configuration knowledge. This conscious design decision is motivated by the concern that minimizing the number of concepts in a modeling language should not compromise the clarity of configuration models. Higher level concepts for representing typical forms of configuration knowledge result in a more compact and understandable representation of a configuration model.

The ontology is by no means the first to synthesize some of these conceptualizations. However, only the conceptualization in (Felfernig et al. 1999) is comparable in that it has combined all the concepts (see Section 3.5 for more details). Partial syntheses of the conceptual approaches include at least the following: in the conceptualization by (Mittal and Frayman 1989), a limited form of functional interactions accompanies the connection-oriented concepts. Clarke (Clarke 1989) presented an approach that extended the function-based approach and combined it with the structure-based approach. A set of concepts combining some aspects of the connection- and structure-based approaches was presented in the configuration design ontology (Gruber et al. 1996). An independent approach also combining product structures and connections was presented by (Axling and Haridi 1994).

Contrary to many previous approaches, the main concepts in this work are treated uniformly with respect to several criteria. They are all defined both as types and individuals to explicitly separate configuration model knowledge and configuration solution knowledge. The main concept types can all be organized in classification taxonomies and have attribute definitions. They can be specified as abstract or concrete to distinguish between accurate and inaccurate information.

Organizing the different concepts in a taxonomy to explicitly represent their common properties and to reduce the maintenance effort is a general idea in configuration design (e.g. (Cunis et al. 1989), (Searls and Norton 1990), (Jüngst and Heinrich 1998)). The explicit distinction between abstract and concrete component types has been made by others (e.g. (Kramer 1991) and (Axling and Haridi 1994)), but (Weida 1996) was the first to consider it a more generic mechanism which applies to other concepts as well, similar to the approach in this work.

In general, the earlier work on modeling structure in configuration knowledge has only modeled the direct has part-relation between a component individual and its immediate parts (e.g. (Cunis et al. 1989; Searls and Norton 1990; Peltonen et al. 1994; Axling and Haridi 1994)). It has not explicitly considered all the semantic restrictions that representing a has-part-relation properly requires (e.g. (Artale et al. 1996)). The conceptualization of resource interactions in the generalized ontology is an extension of the model presented by (Jüngst and Heinrich 1998). The connection related concepts are similar to those presented in (Mittal and Frayman 1989). Integrating the port concept to the structure-oriented concepts, mentioned by (Mittal and Frayman 1989), is explicitly dealt with. In (Gruber et al. 1996) the connections are restricted to the component individuals that are direct or transitive parts of the same component individual. This modeling restriction is not enforced in the generalized ontology.

Most of the research on configuration design has not explicitly modeled the functional domain in the sense presented in this work. The conceptualization of the functional knowledge in the generalized ontology is similar to those presented by Clarke (1989) and Pernler and Leitgeb (1996). However, it has also been argued that there is no clear distinction between components and functions (e.g. (Gruber et al. 1996; Schreiber and Wielinga 1997)). In contrast, in this work these two concepts are kept separate, as the sales persons and customers can view the product through its functions rather than the technical structure of the product.

Constraints have been used in almost all of the work on product configuration to represent the dependencies between components. In the generalized ontology, a constraint is conceptualized as both an object and an expression, similarly to the conceptualization in (Gruber et al. 1996). However, in that approach the constraints are defined as a restricted subset of the underlying KIF language, whereas in the generalized ontology no commitment is made to a particular language or particular forms of constraints for representing them. Introducing constraint sets for representing different points of view on a valid configuration has its roots in observations on how configurable products are managed by companies.

3.1.2 Groundedness

The semantics of all the languages in this work capture formally the idea that a correct configuration must not contain anything that cannot be grounded in the configuration model. The formal definition of this property of configuration knowledge representation languages has been largely neglected. However, it has been identified as important in, for example, the research on dynamic constraint satisfaction problems (DCSP) (Mittal and Falkenhainer 1990) and generative constraint satisfaction problems (GCSP) (Stumptner et al. 1998). In these approaches, the goal is to capture the knowledge that some choices need only be made if other choices have also been made appropriately. In the rule languages of this work, groundedness is uniformly incorporated in the formal semantics of the rules. This distinguishes the languages from, for instance, DCSP and GCSP in which some constructs (e.g. activity and resource constraints) have a grounded interpretation, while others do not (e.g. compatibility constraints).

There is, however, also a more *general groundedness principle* of configuration knowledge representation: anything that is not allowed by the configuration model cannot hold in a correct configuration. Capturing the groundedness in the semantics of the languages allows a more compact and modular representation of such configuration knowledge. Thus, there is no need to add so called "completion" or "frame" axioms that forbid any other state of affairs from holding in a configuration other than those allowed by the configuration model. This leads to a more compact formalization of the configuration knowledge. It also means that if a part of a configuration model or the formalization of the conceptualization is changed, only those parts (e.g. rules) that represent the changed part need to be changed to capture the new configuration model or formalization.

This general groundedness principle is an instance of similar ideas presented in the context of common sense reasoning, non-monotonic reasoning and logic programs. Much of this research has concentrated on formally restricting the set of possible world states, i.e. models, described by a theory to only the *intended models*. The intuition is that if a theory does not speak of some state of affairs, then that state of affairs should not hold in the intended models. This is in contrast to propositional or predicate logic. They allow models with other things in addition to those required by the theory.

Capturing groundedness using classical propositional or predicate logic or CSP is not straightforward. In effect, a semantics with some form of groundedness principle allows more inferences to be made from the same theory than a semantics without a groundedness principle. For example, if the configuration model of a computer were expressed using "standard" predicate logic without the groundedness principle, it would allow models, i.e. correct configurations, where all kinds of other things like the components of a car are included. Using a language whose semantics includes groundedness, like **DWCRL**, allows one to infer that there cannot be car components in a configuration of a computer.

For example, **CRL** cannot be modularly represented by classical propositional logic (Soininen and Niemelä 1998). This means that when changing, for example, a part definition in a configuration model represented as propositional logic, it is not enough to rewrite the one sentence representing that part definition. In addition, the global completion axioms related to part definitions need to be rewritten. It may be possible to restrict the changes to some subset of the entire representation, but in the worst case, the entire representation could be rewritten because of a small local change.

Another semantical issue related to groundedness is the restriction to configurations that are Herbrand models for **FOWCRL** and **DWCRL**. This restriction means that only the objects and their relations mentioned in the configuration model can appear in a configuration. Would then either of these restrictions, groundedness or restriction to Herbrand models, be enough on its own? The answer is no.

Consider the case where only the restriction to Herbrand models is adopted and groundedness is omitted. When representing for example a part definition of a component type, it would not be enough to state the rules in Section 2.7.2 for each part definition. In addition, rules forbidding anything else to be a part of the component with any other part name would also be needed. In addition, there should be "global" rules that only those component types defining parts may have parts. This is because if such rules were omitted, there would be a Herbrand model of the configuration model where such unintended part relations hold. On the other hand, assuming groundedness but not the restriction to Herbrand models would allow a correct configuration with any object, since the set of objects that may appear in a configuration would be arbitrary. Thus, both restrictions are needed.

3.1.3 Approaches to Formalization

Classes of Formalizations

The approach taken in this work facilitates formal representation of configuration knowledge based on different and unified conceptualizations. This is accomplished by capturing the configuration knowledge using a neutral representation language, i.e. **DWCRL**, and implementing the configuration task using an 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. Thus, exploring different conceptualizations of configuration knowledge is supported. Note that rules are not primarily intended to be used by product modelers but as an intermediate language that a higher level configuration modeling language committing to some configuration ontology is translated to.

This is in contrast to the class of previous formal approaches that have fixed a certain conceptualization and then analyzed it and developed special purpose algorithms for the conceptualization. These approaches include the formulation as generic constraint satisfaction problem (GCSP) (Stumptner et al. 1998) and the model in (Najman and Stein 1992)). The Unified Modeling Language (UML)-based approach (Felfernig et al. 1999) commits underneath to the conceptualization of GCSP but also formalizes other concepts, and thus is somewhat similar to this work.

A second class of formalizations of configuration knowledge is based on a neutral general-purpose representation language. This class includes the Configuration Design Ontologies (Gruber et al. 1996), the formalization of the structured oriented PLAKON model (Cunis et al. 1989) of configuration using description logic (Schröder et al. 1996), and also partially the UML and GCSP-based approach (Felfernig et al. 1999). In contrast to these approaches, the **DWCRL** has been specifically designed to allow the representation of knowledge on different aspects of configuration knowledge, types, individuals, constraints and choices, in a uniform and simple manner. At the same time, the computational efficiency of the configuration task has also been considered, which is not the case for the other approaches.

A third class of formalizations differs from the approach in this work in that they do not provide a detailed ontology of configuration knowledge. Instead, they only specify very generic concepts like classes, relations, constraints, individuals, and so on, by using which the configuration knowledge can be modeled. Several DL-based formalizations fall under this category, such as the Constructive Problem Solving (CPS) approach (Klein 1996) and the model in (Owsnicki-Klewe 1988).

Effect of Groundedness on Formalization

A characteristic that separates the formalization in this work from almost all other approaches is that the language used for the formalization has the groundedness property built into its semantics, both by its use of Herbrand interpretations and their groundedness. This makes it much easier to capture in the formalization of the different concepts the notion that a configuration must not contain anything that is not explicitly mentioned in the configuration model. It also simplifies the formalization (see especially Section 3.5).

A further aspect of groundedness is that it clearly distinguishes between the roles of the configuration model and requirements. For the approaches without the groundedness property, one must often resort to meta-level conditions such as that the configuration must not include other concepts than in a configuration model even if the requirements state their existence. Another way to accomplish this is to insist that the requirements can only use a distinct restricted vocabulary dependent on the configuration model. By separating the notions of a configuration satisfying a set of sentences and being grounded by the set of sentences, one obtains the following clear distinction: a suitable configuration must be correct, i.e. satisfy and be grounded by a configuration model, and *satisfy* the requirements. Thus, any form of requirements can be posed without them adding unwanted elements into a configuration.

It is also relatively easy to change the formalization presented in this work due to the groundedness property. For example, adding a similar restriction as in (Gruber et al. 1996) requiring that only component individuals that are direct parts of the same component individual can be connected can be accomplished by simply adding one rule. There is no need to change any other ontological definition or a configuration model represented according to the ontology due to the modularity of representation. This is one advantage of using a neutral representation language not committed to a specific ontology, as well.

3.1.4 Computational Complexity

In light of the many different approaches discussed in this section, it is clear that there is no consensus on the form of the configuration task. This is also reflected in the few available computational complexity results. As a rough characterization, the approaches can be divided to three classes according to the complexity class where the configuration task is or should be: in **NP**, decidable, or undecidable.

*In **NP***

The intuition why configuration tasks should be in **NP** is argued in (Bürckert et al. 1996) based on the properties of problems in that class. It should be easy (i.e. tractable) to check that a given configuration is correct with respect to a configuration model. In addition, a configuration should not be more than polynomially larger than the configuration model. These properties imply that the checking can be done efficiently, which is important for practical tasks. If this is not the case, the formal model may become irrelevant for practical purposes. This characterization reflects the intuition that checking whether a given dependency in a configuration model holds in a configuration is not difficult, but it is the task of combining an appropriate set of elements that satisfies all the dependencies that is the problem. The DCSP, EDCSP, **CRL**, **WCRL** languages and the formalization of the simplified configuration ontology in this work are in this class. However, it is not claimed that they are general enough to cover all aspects. Other approaches in this category include the models of Najman and Stein (Najman and Stein 1992) and those using CSP (Mackworth 1977).

In order to keep the complexity of configuration tasks in **NP**, a semantics with the groundedness condition must be carefully designed. In this work, this is accomplished using a fix point condition that can be checked efficiently. Another approach to groundedness would be to define a subset minimality condition on configurations, as is the case for DCSP (Mittal and Falkenhainer 1990) and consistency-based configuration (Friedrich and Stumptner 1999). However, this is a stricter condition than the one provided in this work. In addition, it may easily lead to a significantly higher complexity, as is demonstrated in the case of rules with disjunctions in the head or EDCSP with minimality condition: the complexity increases to $\Sigma_2\mathbf{P}$-hard (II). This intuitively means that even if one could decide if there is a (possibly non-minimal) configuration in one step of computation, one would still have an **NP**-hard problem in checking that it is the minimal one (Papadimitriou 1994).

In the formalization of the simplified ontology, the assumptions that a storage of individuals that can be in a configuration is given and that the number of variables in the rules of configuration a model and requirements is restricted makes the configuration task remain in **NP**. The first assumption imposes a bound on the configuration similarly to consistency-based configuration (Friedrich and Stumptner 1999). The second restriction in effect restricts the constraints in the configuration model and requirements to n-ary ones, where n is a constant. This is often also the case for CSP approaches, where for instance only binary constraints are allowed. The underlying motivation is to ensure that checking the constraints is easy. Other restrictions with the same effect probably exist as well.

Decidable

In the second class of approaches, the complexity increases beyond **NP** but remains decidable. The increase in complexity can in principle be a result of two factors: the configuration may be very large, or the semantics of the constructs of the configuration modeling language may cause this. In the first case, each definition in the configuration model is easy to check by itself, but the configuration is very large and therefore the checking hard. In the latter case, the constructs of the languages are such that it is hard to check even in isolation whether they hold in a configuration. This class includes the **DRWCRL** and **FOWCRL** languages of this work, the unrestricted consistency-based configuration (Stumptner et al. 1998) and the UML-based approach (Felfernig et al. 1999), which all allow large configurations. There are two further variants in this class: the first assumes that the configurations may be exponentially large (e.g. **DRWCRL**) The second assumes that the size is in principle unlimited *a priori*, but some form of a size limit is imposed for each configuration task instance (Stumptner et al. 1998). A distinguishing feature of **DRWCRL** is that the configuration task does remain decidable although a limit is not given. The approach in this work thus offers a trade-off between expressiveness and implementability. This has not been shown for consistency-based configuration (Stumptner et al. 1998), for instance.

For the approaches that allow large configurations of the first type, it is possible to model the problem using an expressive language and compile it off-line to a (in the worst case exponentially) larger problem representation. The configuration task would then be in **NP** with respect to the compiled representation. This of course does not help for bigger problems if the size of the compilation does increase as the worst case indicates. However, for **DRWCRL** the size of the compilation is dependent on how efficiently the domain predicates divide the set of object constants to smaller domains. If the domain contains many types and there can only be a few individuals of each type in a configuration, such a compilation could be feasible.

Undecidable

The third class of approaches tries to cover a very broad spectrum of problems with highly expressive constructs and allows even infinite configurations. This class includes the CPS approach (Klein 1996), and the PLAKON-based model in (Schröder et al. 1996). However, the generality of the languages in these approaches implies an undecidable configuration task. Therefore, the languages need to be restricted to provide practical systems. It may also be that the configuration checking task remains decidable and can be supported by a system based on an unrestricted language.

3.2 Models of Najman and Stein

In (Najman and Stein 1992) formal models for resource-based and structure-based configuration are defined. The models are given as mathematical structures using set theory. The first model includes a finite set of *objects*, similar to component types, several *items* (i.e. individuals) of the objects which may appear in a configuration, a set of *functionalities* and their *addition* and *test operators*, *properties* of components, and a set of *demands* given in terms of *functions*. The properties define the functions the components produce. The addition operators define how functions are composed. A definition of a solution to a configuration problem is given. Such a solution must be composed of the objects in the configuration model and satisfy the demand. The second model adds to the first a type of structure-based configuration knowledge. The structure is represented using rules similar to **PWCRL** with only cardinality constraints.

The models differ from the approach taken in this work in that they define a mathematical structure, corresponding to a formal ontology, upon which the analysis is based. The models do not define a knowledge representation language in which to formalize configuration knowledge. The conceptualizations in these models are also significantly poorer, as they do not support the definition of, for instance, compositional structures where component types can occur in part definitions of several component types, or the connections of components via ports. Functions in the models and the resource concept in this work are closely related. The formalization of the simplified ontology in this work adds more concepts while retaining the same computational complexity of the configuration task.

The definition of a configuration as a structure ensures that only those objects that are mentioned in a configuration model can appear in a configuration. Thus, a form of groundedness is present in the first model. However, for the second model, the rules are given a semantics close to propositional logic, and thus correct configurations may contain elements that are not grounded by the rules. This is a second significant difference.

The computational complexity of the configuration tasks based on both models is analyzed and found to be **NP**-complete, similar to the approach in this work based on the simplified ontology and **DRWCRL** with the additional assumptions. No implementation or results on the efficiency and practical relevance are reported.

3.3 Description Logic-based Approaches

3.3.1 Description Logics in Configuration

Several approaches to configuration knowledge representation and reasoning are based on different variants of *description logics* (DL) (McGuinness and Wright 1998; McGuinness and Wright 1998; Weida 1996; Buchheit et al. 1995; Klein 1996; Klein et al. 1994). Such logics were developed for representing and reasoning on concept descriptions. Historically they are derived from frame-based languages, semantic networks and other similar representation methods. They include constructs for representing *concepts*, roughly corresponding to classes or types of objects, their *roles*, corresponding to binary relations, *restrictions on the fillers* of those roles, corresponding to definitions of complex binary relations, *individuals*, corresponding to objects, and *roles of individuals*. The main inference services usually provided for a DL are *subsumption*, *classification* and *consistency*. Subsumption is the task of finding if a concept or individual is a special case of another concept. Classification infers where, in the subsumption hierarchy of concepts, a concept or individual belongs. Consistency checks if a set of concepts and their role definitions are consistent or if a set of individuals and their roles are consistent with respect to their concept definitions.

Such logics provide tools for the knowledge engineering task, i.e. building a conceptual model of the domain by defining concepts (e.g. components), their roles and so on. This is the main difference to the approach taken in this work. Here, the configuration models and the classification hierarchy are assumed to be correct, complete and known, since the emphasis is on the configuration task. It has also been argued that at least in some technical domains the concepts and their subsumption relations are usually rather well known, so such services are not crucial (Friedrich and Stumptner 1999).

However, a DL-based system component can also be used to solve a configuration problem. The main services of the DL component in solving the problem are the classification of an individual to the most specific concept on the basis of its properties, inferring additional properties that it (and other individuals) must have for the configuration to be correct on the basis of the concept descriptions, and checking that a configuration is consistent with respect to the concept descriptions. These services are particularly suited for incremental configuration where incomplete information is given and the individuals need to be refined when more information becomes available.

Despite the utility of these reasoning services, DL-based approaches usually also extend the basic formalism with some sort of general forward chaining rules or constraints (McGuinness and Wright 1998; Klein 1996). This seems to be due to two reasons.

First, the basic DL semantic used in these approaches is based on an *open world assumption*, similar to classical logics. This means that it cannot be inferred if all the things in a configuration have been specified accurately enough, i.e., if the configuration is complete. As the open world semantics corresponds to the classical semantics of, for instance, predicate logic, it also means that the basic DL semantics lacks the groundedness property. This problem has also been raised in (Schröder et al. 1996) and addressed more thoroughly in (Weida 1996) by using the *closed terminology assumption*. It essentially states that only individuals of those concepts that are in a configuration model can occur in a configuration, that they may be related by only those roles defined by their concepts, and that every individual in a configuration corresponds to some concept in a configuration model. This provides a definition for completeness of a configuration. It also increases the set of inferences that can be made from the concept descriptions. Note that the definition of the simplified ontology in Section 2.2 and its formalization using a language with the groundedness property in Section 2.7 in effect enforce the closed terminology assumption.

Second, concepts, roles and their subsumption hierarchy are not enough to express typical configuration constraints that may refer to more than two concepts or concepts that have no relation in terms of roles (McGuinness and

Wright 1998; Klein 1996; Schröder et al. 1996). For instance, the DL language of the CLASSIC system was found to be lacking in configuration problem solving when rules or constraints needed to be represented (Bürckert et al. 1996). In contrast to this, the approach taken in this work specifically addresses the definition of arbitrary constraints with the groundedness property as EDCSP or rules. In this sense, the approaches based on a DL can be seen as complementary to the approach in this work. Some sort of hybrid representation combining grounded rules and a DL is no doubt feasible. In fact, several formally well founded combinations of a DL and rules have been proposed, for example in (Baader and Hollunder 1994), but not for configuration-specific rules such as in this work.

Although the approaches based on description logics can be seen as complementary, in the approach in this work a mechanism for representing individuals and types, corresponding to concepts, is provided as well. This takes the form of domain predicates ranging over individuals, which can be used to represent the unary predicates corresponding to concepts in a DL. However, this mechanism is not intended for subsumption-like inference. The domain predicates have a fixed extension in a configuration model, corresponding to the intuition that the class hierarchy is known. However, they can be used to restrict the applicability of rules to only individuals of a given type. This approach is similar to introducing *sortal predicates* and *sorted quantifiers* in a simple *many-sorted logic*, most of which assume a fixed extension of sorts as well (Cohn 1989). In a many sorted logic, every variable has an associated *sort*, corresponding to a class or type, determining the objects over which it ranges.

3.3.2 PROSE

A DL-based implemented approach is reported in (McGuinness and Wright 1998). The implementation has been used successfully since 1990 in interactively configuring telecommunications products. The configurator, PROSE, is based on the CLASSIC knowledge representation system. No configuration specific ontology is reported except for the concepts of components, their relationships and forward chaining rules. For the variant of DL used in PROSE the subsumption algorithm is tractable. It cannot therefore capture as hard configuration problems as the languages in this work. The effect of adding the forward chaining rules to the DL on complexity is not analyzed. The reported benefits of the system include reduced order processing time, elimination of errors and rework, reduced staffing and easy maintenance of the configuration models

3.3.3 Constructive Problem Solving

In *Constructive Problem Solving* (*CPS*)(Buchheit et al. 1995; Klein 1996; Klein et al. 1994), a configuration is defined as a (possibly partial) Herbrand model of a theory in a description logic-like language. The language is extended with rule-like first order constraint formulas and concrete domains for numeric attributes and arithmetic on them. The language is also given a formal semantics. The configuration task is formally defined as finding a Herbrand model of the configuration model represented in the language such that the model satisfies a set of goals, i.e. requirements. The task is also shown to be undecidable for the language.

In contrast to the approach in this work, CPS does not commit to any specific conceptualization of configuration domain. The language is significantly more complex, and allows arithmetic on concrete domains unlike the approach in this work. However, it only allows features, i.e. functional roles. These re not adequate for representing part relations (Schröder et al. 1996), for example. The CPS approach does not seem to require that a configuration is grounded in the configuration model, although the restriction to Herbrand models does accomplish this to some extent. The relation and expressive power of the rule-like constraint formulas to the rules in this work is not clear. They are of a more restricted form called definite rules, i.e. rules with no disjunction and negation. On the other hand, the atoms they are composed of are more complex. They may contain existential quantification, for example.

A major distinction between the approaches is that the CPS approach leads to undecidable configuration tasks, whereas the formalization in this work leads to a decidable task. Without restrictions, CPS is a theoretical study that cannot be implemented directly. No results on implementing the approach or its efficiency and practical relevance are available.

3.3.4 Description Logic-based Formalization of the PLAKON model

In (Schröder et al. 1996), a partial logical reconstruction of the PLAKON (also known as KONWERK) configuration system is given. The reconstruction is based on a DL extended with first-order logic sentences for representing n-ary constraints. Again, a configuration is considered a model of the theory defining the configuration model. The configuration knowledge modeling language of PLAKON based on modeling component types, attributes, compositional structure and general constraints is translated to the extended description logic language.

Completion axioms partially enforcing the closed terminology assumption are added. The description logic used contains relatively powerful primitives, although they are used in a restricted manner. Using such primitives may lead to a high computational complexity of the inference tasks for configuration. In fact, some of the constructs may even lead to undecidability (Schröder et al. 1996), which contrasts with the formalization in this work. However, no complexity results are presented for the formalization.

The major differences to the formalization in this work are the lack of groundedness of configurations and the more restricted conceptualization. A configuration in the reconstruction of PLAKON may contain individuals of new concepts defined as a combination of the concepts in the configuration model, or even arbitrary new concepts. This due to incomplete handling of the closed terminology axiom and the open world semantics. However, in (Schröder et al. 1996) it is pointed out that allowing such freedom would be useful in providing a language for a support tool intended for a more innovative type of design.

No implementation or results on efficiency and practical relevance are reported.

3.3.5 Model of Owsnicki-Klewe

In (Owsnicki-Klewe 1988) probably the first formalization of configuration knowledge and task is presented. It uses a variant of DL extended with weak implication rules. This language is noted to be too weak to represent generic constraints in configuration knowledge. A fix point equation characterizing a correct partial configuration is defined. However, the approach is interactive in nature, i.e. the reasoning mechanism for the language does not complete the configuration. The user has to supply some decisions. This is in contrast to the approach taken in this work. On the other hand, the fix point equation does provide a form of groundedness of a partial configuration in this approach. However, it is not required that the result would be the least configuration satisfying the equation, which would enforce a similar groundedness as in this work, although this seems to be implied by the discussion on the equation.

No ontology or complexity results, implementation, or results on the efficiency and practical relevance are provided.

3.4 Configuration Design Ontologies

The most detailed formalization of a configuration knowledge conceptualization is given in the *Configuration Design Ontologies* (Gruber et al. 1996) (*CDO*).

These ontologies were developed in connection to the VT/Sisyphus elevator configuration challenge problem (Schreiber and Birmingham 1996). As such, they may be influenced by domain specific considerations. This is most evident in the ontological commitment to viewing configuration as *parametric design* extended by some aspects of part selection. The problem is thus described using a set of parameters, corresponding to attributes, and constraints on them. This makes the ontologies somewhat method dependent, in contrast to the approach in this work.

The CDO consists of a generic configuration ontology and a VT specific theory. The generic configuration ontology includes the following set of concepts: component types and individuals, structure, connections, parameters and constraints. Component types model the structure by defining their direct parts, similar to the simplified ontology in this work. However, the cardinality is always one, and no alternative possible part types are allowed. Defining subtypes of a part type allows a restricted form of possible part types. Connections are modeled as relations between component individuals, instead of using ports to model the interfaces. The generic connection relation can be specialized which allows the modeling of different types of interfaces. However, since ports are lacking, there is no mechanism for specifying the "places" in a component to which another component is connected. Connections are reified to connection components, to allow the modeling of the properties of connections. Connections between wholes and parts are not allowed.

The configuration design ontologies are formalized using Ontolingua, whose formal semantics is in turn defined using the Knowledge Interchange Format language, a variant of first-order predicate calculus (FOPC) (Gruber 1993). The conceptualization of that ontology is in several ways more limited than the simplified ontology in this work. It also does not provide a language for representing knowledge based on the ontology. As the semantics is based on a classical logic, no groundedness principle is included in the semantics. In several cases the need for some type of closed world assumption is mentioned, but not formally given. There is no assumption that the models of the language are Herbrand models. Therefore, a configuration with respect to a configuration model committing to this ontology without additional restrictions could contain arbitrary objects and relations between them.

No implementation based directly on CDO is reported. However, some implementations developed to answer the Sisyphus challenge at least partially use the ontology, which makes it relevant for the elevator domain.

3.5 UML-based Conceptualization and Consistency-based Configuration

A conceptualization of configuration knowledge combining many of the previous approaches is presented in (Felfernig et al. 1999). In that approach, the static diagrams of the Unified Modeling Language (*UML*) (Fowler 1997) for representing object-oriented analyses and designs is extended with configuration knowledge concepts. This is done through the *stereotype* extension mechanism of UML and uses the *object constraint language* (*OCL*) extension of UML. In addition, the concepts are given a formal semantics within the framework of consistency-based configuration (Friedrich and Stumptner 1999).

3.5.1 Ontology

The UML-based conceptualization includes component types, their aggregation and generalization hierarchies, ports and connections, resources, attributes, generic constraints, and special cases of constraints called requires- and incompatible-relations between component types. Thus, of the concepts in the simplified ontology in this work only functions are missing.

However, there are some differences. The refinement of properties such as part definitions seems to be missing from the UML-based conceptualization. The exclusivity definition of a part definition allows specifying a cardinality for how many wholes a component can be part of in the UML-based conceptualization. A restricted form of context for resource production is introduced in that the producers and users of a resource must be parts of a common component. The ports are modeled as parts of the components that have them, which seems a bit counterintuitive. These differences in the details are relatively minor. Other differences are that the UML-based conceptualization includes attributes, shared parts, and two specific types of constraints. In the development of the conceptualizations relatively similar design decisions were made, which gives some credibility to both.

The visual notations in this and the UML-based approach are different, although the difference is mostly just a very simple syntactical change. The notation in this work is the same as the UML-based notation for types, Isa-relation and structure. The UML-based notation has the benefit of being based on a rather widely used notation for software engineering activities. This might make its use by product modelers easier. On the other hand, all the concepts are modeled using rather similar boxes and lines, without distinguishing visual characteristics, as is the case for the visual notation in this work. Providing a clearly differing notation may make the configuration models more understandable.

3.5.2 Formalization of the UML-based Conceptualization

The formalization in (Felfernig et al. 1999) is carried out using a variant of first order logic within the framework of consistency-based configuration. A mapping from the formalization to a representation in GCSP and solving the configuration problem using an implementation based on GCSP is also discussed.

The formalization of the UML-based conceptualization uses range restricted first order logic clauses extended with restricted sorted existential quantification, some set and aggregate constructs, and interpreted functions. The semantics of the language is not clear, especially for the set and aggregate constructs, unlike in the approach in this work. In addition, the formalization in this work is done directly using the language that provides the implementation, which makes the relation between conceptualization and implementation simpler.

The mapping from the conceptualization to the formalization in (Felfernig et al. 1999) is more complex and results in a larger and more complex set of sentences than in this work. This is partially due to a bit more complex conceptualization, but there are also other significant differences. Due to the groundedness property, the representation in this work is modular in the sense that each ontological definition, type, property definition and constraint can be formalized independently of other things. This is not the case for the UML-based approach. Many of the ontological axioms given as independent sentences in this work are not defined independently in (Felfernig et al. 1999), but with reference to a given configuration model. Therefore, they need to be changed if the configuration model is changed.

The definition of a valid configuration in (Felfernig et al. 1999) does not include the minimality condition, which would enforce a groundedness property. For example, it is necessary to separately add a sentence stating that every component individual must be a part of some whole that may have it as a part according to the configuration model. In the formalization of this work, this additional sentence is not required, since a component individual can be in a configuration only if is introduced by some rule as a part of some whole. The difference means that if the configuration model changes with respect to the part definitions in it, this sentence will need to be rewritten in the UML-based approach. In the rewriting, all the part definitions in the configuration model must be taken globally into account.

Furthermore, the compositional structure of the product is represented in (Felfernig et al. 1999) using ports and connections, which leads to more complexity. The underlying GCSP formalism and its implementation seem to have affected this decision. This is in contrast to the formalization of the simplified ontology in this work, which tries to be implementation and method independent.

Weight constraints and domain predicates allow a compact representation of things that require complex predicates in (Felfernig et al. 1999). As an example, a very complex sentence is needed to enforce the lower bound of a cardinality in a part definition. This is simple using cardinality constraints. Component types are represented as object constants in (Felfernig et al. 1999) unlike in this work. Thus, the relatively simple mechanism of using unary predicates for representing types of objects is not used. This necessitates explicit modeling of the inheritance of properties.

To represent the knowledge that a component individual must be transitively part of the same component individual as some other component individual, *path constructs* are defined in (Felfernig et al. 1999). The path constructs are equivalent to the transitive closure of the has part relation in this work, which can be easily defined in the standard manner using rules. Thus, similar path constructs can be easily incorporated into the formalization in this work.

3.5.3 Consistency-based Configuration

In (Friedrich and Stumptner 1999) it is shown that the configuration task can be considered analogous to the consistency-based fault diagnosis task. Therefore, the results from that domain can be imported to the configuration domain. Only the formalization of configuration as consistency is discussed here.

In consistency-based configuration, a configuration problem is defined as a union of two theories. The first represents a configuration model and the second the requirements. The configuration model theory defines a set of component types, their ports and attributes, the domains of attributes, and constraints. The types, ports, attributes and the values in the domains are all represented as object constants. The formalization is limited to discrete domains.

A configuration is defined to consist of the following three theories:

- a set of ground atoms defining the component individuals in a configuration and of which types in the configuration model they are of
- a set of ground atoms defining the connections between component individuals through ports defined in the configuration model, and

- a set of ground atoms defining the attribute value assignments for the attributes of component individuals defined in the configuration model.

In addition, there are three ontological axioms that are included in every configuration model. These state that each component individual must have a unique type, each port takes part in at most one connection and each attribute must have a unique attribute value.

A configuration is defined as a union of the three theories representing the components, connections and attribute value assignments. A consistent configuration is such that its union together with the configuration model and requirements is satisfiable, i.e. that there is a model of the entire theory thus formed. The consistency of a configuration is not enough for defining correctness, since if the configuration model and requirements together are satisfiable, then the empty configuration is always a consistent configuration.

A further requirement on a correct configuration is that it must be complete, i.e., everything that needs to be chosen has been chosen. This requirement is formalized by adding a set of *completion sentences* to each of the sets of sentences defining the components, ports and attributes. In effect, these represent the restriction that no ground atoms of similar form as in the configuration except those that are explicitly mentioned in the configuration are true. There are no other component individuals of these types, no other connections, and no other attribute value assignments. Now, a *valid* configuration is defined as one whose completed form is consistent with the configuration model and requirements, i.e. that there is a model of the entire theory. Furthermore, an *irreducible* configuration is a valid configuration no subset of which is a configuration.

When comparing consistency-based configuration with the approach taken in this work, the first obvious difference is the notion of a configuration. A configuration is defined as a theory of first order logic in the first and as a Herbrand interpretation of a set of sentences in **DWCRL** in the latter. Although the form of the theory in consistency-based configuration actually is equivalent to a Herbrand interpretation, in the sense that it has a unique Herbrand model with exactly the same contents as the theory, there are three differences. First, in the Herbrand interpretation alternative it is not necessary to restrict the form of the ground atoms in a configuration to use the vocabulary of the configuration model. This is built into the definition of what a Herbrand interpretation is. However, similarly as in the formalization in this work, the component identifiers are separate from the configuration model.

The second, more important, difference is that a model of a set of sentences containing positive information, i.e. that something must exist or that a choice between two alternatives must be made, cannot be empty, like a theory consistent with a set of sentences. A model must contain "enough" facts that every sentence of the theory representing a configuration model becomes true. Therefore, a configuration in the approach in this work is always complete. This may not be desirable if the correctness of *partial configurations* needs to be defined, which is useful for interactive configuration applications where a user incrementally specifies a configuration. For discussion on how to extend the approach in this work to cover partial configurations, see Section 4.2.

The third significant difference to the approach in this work is related to the groundedness property of the representation language. This property eliminates the need for the completion sentences, since that type of negative information is provided by the semantics. In this sense, the valid configurations may contain unnecessary elements when compared to the approach in this work. However, a form of groundedness of configurations is in fact recovered in consistency-based configuration in the definition of an *irreducible configuration* by requiring that a configuration is subset minimal. This form of groundedness is stricter than the form presented in this work and seems to be more related to the optimality of a configuration rather than its correctness. In particular, in (Friedrich and Stumptner 1999) it is suggested that optimality criteria such as cost optimality could be defined on top of the set of valid, irreducible configurations. It is not clear that subset and cost optimality can be thus ordered, as they may be more or less dependent on each other. For example, if a cost optimality criterion were to allow negative costs of component individuals, then a subset minimal solution is not necessarily the optimal solution.

There is also a somewhat harmless anomaly in consistency-based configuration caused by defining the requirements as an additional theory that must be satisfied. Requirements cannot introduce anything that is not mentioned in the configuration model into an irreducible configuration because of the minimality condition. However, consider the following example. Given a configuration model of PC, is there a configuration that satisfies the condition that Cleopatra is the queen of Egypt? If the condition "Cleopatra is the queen of Egypt" were added to the requirements, the answer would be yes, provided that the rest of the requirements together with the configuration model are consistent. This is because the resulting theory would be satisfiable, although the configuration part of the theory would not contain Cleopatra in any role. Thus, the consistency-based approach fails to distinguish between 1) satisfiable or unsatisfiable requirements and 2) requirements that make no sense with respect to a given configuration model.

This anomaly can be corrected in the consistency-based configuration by insisting that the requirements are given using the terminology of the configuration model, i.e. using the constants and predicate symbols in the configuration model. However, in the approach taken in this work, the restriction to Herbrand models and groundedness at least partially eliminates the need for posing additional restrictions on the form of the configuration and the requirements. These restrictions in effect mean that only the requirements posed using the terminology of the configuration model can be satisfied. On the other hand, this formulation is not perfect either, since it fails to distinguish between requirements that cannot be satisfied and requirements that do not make sense. For both these cases, the answer is that there is no such configuration. The distinction between them could be explicated by restricting the requirements to use the terminology of the configuration model.

3.5.4 Generative Constraint Satisfaction Problems

The UML-based conceptualization is first translated to a variant of first order logic within the framework of consistency-based configuration and then to GCSP to provide an implementation. Therefore, GCSP is discussed in some detail in this section to give a complete account of the UML-based approach and to contrast it with the approach taken in this work.

GCSP represents configuration problems by explicitly modeling *component*, *property* and *port* variables (Stumptner et al. 1998). It also allows an infinite, non-predefined set of such variables in a solution. GCSP partially extends DCSP with *generic constraints* and *resource constraints*. However, only a restricted form of activity constraints is allowed.

Generic constraints allow the writing of constraints over undefined sets of variables by incorporating meta-variables that refer to the CSP variables active in an assignment. The constraints have a similar form as the inclusive choice rules of **CRL**. However, instead of propositional atoms they are constructed of the meta-variables and decidable predicates. These predicates may refer to the values that variables have in an assignment, the identity of variables and the activity of variables. In addition, resource constraints may be stated. These are aggregate functions on intensionally defined sets of variables and may be used in the generic constraints as well

The activity constraints are used to introduce the property and port variables representing the properties and ports that a component variable of a given type has. This generates active variables to a solution. New active variables are also generated when a resource constraint indicates that there is a need for additional resource production. The third case for variable generation is when a port variable must be instantiated with a component variable to represent a connection, and an applicable component variable is not active in the assignment. These three cases account for the dynamic change in the set of active variables. Note that the third case reveals that component individuals are represented both as variables and values.

There are three major differences between GCSP and the approach in this work. The first is related to the treatment of the semantics. In (Stumptner et al. 1998) the semantics is explained in detail and is defined to include a kind of groundedness condition, explicitly for property variables and implied for other variables as well, as they are only generated in the above three cases. However, GCSP has no formal semantics that would capture this groundedness aspect. This distinguishes it from the approach taken in this work. On the other hand, EDCSP in this work allows only cardinality constraints on the activity of variables, but no resource constraints using aggregates over variables. In addition, EDCSP has a finite, predefined set of variables. It would seem that extending EDCSP with infinite sets of variables, meta-variables and resource constraints is possible. Thus, such an extension could be used to give a formal account of GCSP as well.

The second major difference is the one between GCSP and **DRWCRL** in the treatment of the component types and component individuals. In GCSP, the component individuals are CSP variables whose domain is the set of component types. A component individual is of the type that the corresponding variable has as a value. In contrast, in this work object constants, corresponding to values of CSP variables, represent the individuals, while unary predicates, corresponding to CSP variables, define the type of the individual. This allows usage of the standard notions of quantifiers and variables in rules without introducing meta-variables and the dual representation of component individuals as variables and values. However, the current semantics of **DRWCRL** does not either allow a non-predefined, infinite set of individuals.

The third major difference between **DRWCRL** and a constraint-based approach is the difficulty of representing relations such as parthood or connections. This is facilitated by the predicate symbols in **DRWCRL**. In a constraint-based approach, one has to encode relations in the variable names. For example, property variables are encoded with names of form $t.p$ when type t has the property p in GCSP. Another example is encoding the connections between components by the port variable having as a value the name of a component variable. This representation problem motivated using **DWCRL** instead of EDCSP or the propositional rule languages for formalizing configuration knowledge in this work.

Similar to GCSP, a form of resource constraints is included in the rule languages in this work in the form of weight constraints. However, GCSP allows a richer set of aggregation functions than just the sum in this work (e.g., maximum value). In addition, constraints using arithmetic are included, which are missing from the languages in this work. Finally, in the constraint-based approaches there is no mechanism for defining predicates using other predicates, which is facilitated by rules. In addition, there are no set-valued variables. This means that, for example, it is very difficult to define the transitive closure of the has-part relation using constraints.

3.5.5 Computational Complexity

The computational complexity of the configuration task within the framework of consistency-based configuration or as a GCSP is not analyzed, unlike in this work. However, it is noted that unless restrictions are made in the depth of the terms in the formalization of consistency based configuration, infinite configurations are also allowed and thus undecidability could result. This is considered undesirable and simple restrictions based on the number of component individuals in a configuration are proposed. On the other hand, a restricted form of quantification is proposed to allow the representation of configuration models for which the configurations do not have a predefined size limit.

3.5.6 Implementation

There is no report on an implementation of the UML-based conceptualization or consistency-based configuration. However, a related configurator tool called COCOS implements a configuration modeling language LCON that is close to the UML-based conceptualization (Stumptner et al. 1998). LCON is a frame-like language that supports modeling component types, their taxonomy, connections, structure and constraints. LCON is also closely related to GCSP. In the COCOS implementation, LCON is translated to GCSP and the configuration problem solving is based on the GCSP formalism. The structure related model elements are translated to connections in a way similar to the UML-based approach.

The COCOS tool supports a fully automatic configuration mode, checking and completion of configurations and an interactive configuration mode. A tool called LAVA based on COCOS was reported in 1998 to have been in successful operative use for two years. It was used for configuring digital switching systems with as many as tens of thousands of components. The main reported benefits are that it improved the quality of configuration results, was considerably less expensive to implement than the approaches tried earlier in the company, had a better functionality than the earlier approaches, and was more maintainable (Fleischandelr et al. 1998).

4 Further Work

In this section, further empirical and theoretical research needed for concretizing the approach taken in this work is discussed.

The practical relevance of the ontology and the prototype implementation of the languages gained credibility based on a few small examples. However, more work is needed to empirically validate the ontology, the languages and scalability of the implementation by testing them on a larger sample of products and larger products. This requires implementing a configurator based on the approach taken here. Modeling large enough samples and products is very time consuming but crucial for validating this and other approaches to configuration. Alternative approaches to generating large random test sets are discussed. The main problem identified with this approach is that it may be very difficult to develop a practically relevant statistical model of products in general.

The implications of the complexity results are also briefly discussed. The main thrust is that the configuration task seems to be at least **NP**-complete but there are several reasons why the worst case behavior may not actually be relevant for real products.

In addition to empirical work, there is a clear need for further formal work in developing and analyzing the ontology and languages. This work is needed to bring this approach closer to practice. The further work includes incorporating geometric, pricing, scheduling, optimality and control knowledge into the ontology and arithmetic into the languages. Further on, it is necessary to extend the semantics of the languages to handle partial, incomplete configurations for interactive configuration where the configurator supports a human. In addition, problems with no pre-defined bounds on the size of the configuration may need to be supported. Finally, a proper visual configuration modeling language based on the ontology in this work should be defined. It should also have a direct model theoretic semantics instead of translating it to another language.

4.1 Preliminary Evidence for the Approach

The empirical and constructive part of this work is not extensive enough to conclude that the approach can be used for practical configuration tasks. These parts were given less emphasis, since the intention was to explore the approach and provide some proof of concept before proceeding with more extensive empirical work. However, there seems to be enough evidence of practical feasibility that the approach is worth pursuing further. This evidence is discussed next for the ontological part and implementation of the formalization.

4.1.1 Configuration Ontology

The generalized ontology covered the modeling needs of the three case products well. Part definitions were found a convenient way of representing the example products. The compositional structures were easy to construct. This is also the way that product developers describe products (Tiihonen et al. 1998). Ports and connections would probably be easier to identify and model than they were if the interfaces of the component types in the product were originally set to a high priority and thus better documented. Resource interactions were not so evident in the rock drilling equipment, whereas in the PC and hospital monitor they were easy to identify. The examples required relatively few general constraints. Thus, for these examples the generalized configuration ontology covered the typical configuration phenomena well. In fact, the simplified ontology with few extensions (e.g. attributes) would also have been equally applicable.

4.1.2 Formalization and Implementation

The formalization of the simplified configuration ontology was easy using **DRWCRL**. The efficiency of the prototype implementation for the languages is sufficient for the small set of small problems that it was tested on. However, these problems were very restricted in size and the phenomena they captured. Therefore, it is not clear that the approach scales up to large products or different products. This is in contrast to the approaches for which empirical evidence for solving real configuration problems exists (McGuinness and Wright 1998; Fleischandelr et al. 1998; Yu and Skovgaard 1998).

However, there are indications that the proposed formalization and its implementation provide a basis for solving real product configuration problems. Experience in other domains such as planning, satisfiability checking, model checking and Hamiltonian circuits have shown that the implementation of the basic constraint rules on which the prototypes are based is capable of handling large problems. It also solves such complex problems with a comparable or in some cases considerably better performance than the best problem solvers in these domains (I, (Simons 2000).

A formalism with the groundedness principle allows a more compact representation of knowledge than a formalism without such a principle. A more compact representation may also beneficially affect the efficiency of problem solving. In (Simons 2000), it is empirically shown that for at least some problem domains whose representation benefits from the groundedness principle, the efficiency of problem solving is better than when using a formalism with no groundedness principle. The comparison is carried out between the Smodels system implementing basic constraint rules and several state-of-the-art implementations of the 3-SAT formalism.

More compelling evidence for the approach in this work is provided in (Syrjänen 1999). In that approach, a rule language is used for formalizing the configuration model of an operating system. The ontological basis of that approach is simpler. It consists of objects and their simple dependencies. In addition, the rule language is simpler than **DRWCRL**. The rule language is given a formal semantics that incorporates a similar groundedness property as in this work. The complexity of the configuration task is shown to be **NP**-complete. The implementation is based on the same system and form of basic constraint rules as in this work. For this configuration model, the system seems to provide adequate performance for practical purposes, i.e. running times of a few seconds. The configuration model has roughly two thousand components, which results in a set of tens of thousand rules in the representation.

4.2 Further Empirical Evaluation

There is a clear need for more empirical research to validate the feasibility of the ontology, formalization and implementation in this work. Such empirical research in the form of modeling and solving real configuration problems would also provide input for theoretical work.

4.2.1 Validating the Ontology

The generalized ontology should be validated by modeling a larger set of different kinds of products. It may also have to be extended or changed. The value of the ontology depends mostly on its utility in modeling different products. In addition, a more comprehensive set of modeling guidelines should be developed to provide a modeling methodology. Such empirical evaluation is time consuming and requires access to the confidential information of many companies. However, this seems to be the only way to properly validate the ontology.

4.2.2 Testing the Implementation and Formalization

As with the ontology, the implementation based on the formalization should be tested on a larger set of larger problems to show whether it scales up and is practically useful in supporting configuration tasks.

Before empirical evaluation, one needs to consider the following question: As configuration tasks are **NP**-hard, i.e. intractable, for most models of configuration, can efficient support for configuration tasks be offered by a computer? The answer may be "yes" for two reasons. First, the analysis is based on worst case complexity. It may be that configuration problems are so well structured that the exponential worst-case behavior implied by **NP**-completeness does not materialize. This structure could be the result of the product being designed by humans. Designers very often recursively decompose a design problem into relatively independent parts. Thus, a product is typically not an ill-structured system where everything depends on everything else.

Second, there are several reports on the successful use of systems based on intractable approaches. Heuristic control knowledge in the COCOS system (Stumptner et al. 1998) and the interactive configuration mode in the PROSE system (McGuinness and Wright 1998) provided efficient systems that have been in operational use for configuring very large products. In other cases, the products have been configured efficiently enough with complete search algorithms for propositional logic and rule-based representations without problem specific heuristics (Yu and Skovgaard 1998; Syrjänen 1999).

Acquiring access to large enough sets of product models to get statistically meaningful test results is difficult, since the models are often confidential. As noted above, it is also very time consuming to model large products. Another approach to providing empirical results is to use large randomly generated problem sets. This has the failing that, as suggested above, real products may exhibit particular structural features and not follow a random distribution.

One way of trying to work around these difficulties is to combine the best of both worlds. This could be accomplished by modeling some set of real world products and analyzing their common and varying characteristics. A statistical model based on the analysis could then be developed. Using the model, large random problem sets could be generated for testing purposes. This idea is similar to the idea underlying the GraphBase system. GraphBase contains graphs representing real world phenomena such as maps. Using these as a seed, it can generate large random graphs for testing algorithms (Knuth 1994).

4.2.3 Constructing a Configurator

In order to proceed with large-scale empirical tests of the ontology and implementation, it would be necessary to construct a system for modeling configuration knowledge and solving configuration problems. Such a system should have a good graphical modeling environment to facilitate fast modeling. It should implement the translation from the configuration modeling language to rules and allow the use of the implementation for configuring the products. Of course, it should be instrumented to allow testing on large sets of configuration models and to provide measurement data on the performance.

For such a system, simplicity of the representation and understandability of configuration models is a challenge. A computer-based tool at least in principle allows a clear graphic representation of configuration models. However, even this may not be enough if the models become very large. Additionally, hierarchical browsing of configuration models, for instance based on the product structure, and separate views for classification, part definitions, ports, resources, functions and constraints would enhance the usefulness of the system.

4.3 Further Theoretical Work

4.3.1 Ontology

Like the previous approaches, the ontologies presented in this work are probably partial ones. A more complete ontology would probably include geometric, pricing, scheduling and optimality related knowledge and the knowledge on how to configure a product. The geometric layout of a product is important for some products (Tiihonen et al. 1996) and may influence configuration decisions based on, for example, space constraints. For practical tasks, it is also not enough to generate only the description of a correct product. A configurator also needs to model and reason on the price or cost of the product, and on when it can be delivered to the customer, i.e. on scheduling knowledge.

Furthermore, not all requirements are strict and sometimes there may be several suitable configurations for a given set of requirements (Stumptner et al. 1998). Therefore, one should be able to model and reason on preference or optimality criteria and *soft constraints and requirements*. These types of constraints and requirements are such that it would be nice if the configuration satisfies them but this is not necessary. Finally, some configuration tasks may be such that they cannot be carried out efficiently using a general complete algorithm.

Thus, it may be necessary to extend the ontology to cover control knowledge, i.e. knowledge on how to configure the product. This approach has been taken in, for example, the COCOS system (Fleischandelr et al. 1998).

4.3.2 Extending the Formalisms

While the formalisms in this work, particularly **DRWCRL**, provide a straightforward representation of many aspects of configuration knowledge, some extensions to their syntax and semantics would make them more useful. In addition, potential extensions to the ontology and alternative definitions of the configuration task may require new constructs. The most important of these extensions are real and integer arithmetic, aggregate functions, optimization criteria, partial configurations, control knowledge and support for configuration tasks with unlimited configuration size.

Arithmetic on integers and reals is an important representation format for many forms of engineering knowledge, such as geometric knowledge on dimensions or physical measures such as weights of components, or pricing and scheduling related knowledge. The formalisms in this work should be extended with constructs allowing arithmetic to facilitate representing such knowledge. Additionally, aggregate constructs similar to those introduced in, for instance, Datalog and other database languages would probably prove useful for expressing general global constraints over the configuration. Arithmetic constraints and aggregates are supported by the COCOS system (Stumptner et al. 1998)

Since incorporating arithmetic easily leads to undecidable configuration tasks, the expressivity of such an extension needs to be carefully balanced against the complexity of configuration. The first candidates would be such that their decision problems do not increase the complexity beyond **NP**, for instance integer programs (Papadimitriou 1994). A natural extension of arithmetic and weight constraints is to provide constructs to define optimization criteria and an implementation that allows optimization based on those criteria.

Another direction to pursue is to define semantics for the languages in this work that would formalize the notion of a partial, incomplete configuration. Such a semantics would allow extending the formal model to cover interactive configuration, where the user makes the hard decisions and the computer only the efficiently computable ones. For example, the propositional logic-based system in (Yu and Skovgaard 1998), the PROSE system (McGuinness and Wright 1998) and the COCOS system (Stumptner et al. 1998) all provide such a configuration mode. This may be the only feasible alternative for very large or complex problems. In addition, this type of assistance in configuration tasks can be more acceptable to companies than a completely automatic configurator (Tiihonen et al. 1996).

In interactive configuration, a user would specify a partial configuration. After this, the configurator would compute the immediate consequences of the partial configuration with respect to a configuration model, thus possibly adding or removing some elements. Then, the user would add some specifications, and so on. In this configuration mode, it is very important that the partial configuration can be computed efficiently. A further line of research also supporting the configuring of large products would be to investigate how to incorporate control knowledge in a formally well-founded manner to the languages in this work.

Some configuration problems may arguably be such that no limit on the size of the configuration can be defined before solving the problem instance. As an example, configuring a telecommunications network based on very abstract functional requirements may give no indication of a size limit on the number of nodes within the network. A system implementing a restricted form of such problems and in operational use is described in (Fleischandelr et al. 1998). However, since a product is usually a physical one or embedded in a physical framework, such as software in a computer, a configuration in principle should be finite.

There are at least two different possibilities to extend the approach in this work to cover configuration problems without pre-defined bounds. The first is to follow the approach taken by the GCSP by introducing "generators" in the formalism that in effect make the set of available component individuals infinite. Adding such generator functions to the language of **DRWCRL** or **FOWCRL** makes the Herbrand Universe of a set of rules infinite. The second is to allow an infinite set of rules without function symbols. For the formalization of the simplified ontology, this would mean that the storage of individuals would be infinite. Both extensions accomplish the desired results. However, the first would allow a finite representation of the set of infinite individuals, which makes it more amenable to practical implementation. Unfortunately, this does not mean that decidability would be preserved in this option. Already allowing functions in the subset of **FOWCRL** corresponding to normal logic programs with stable model semantics would lead to highly undecidable configuration problems. This can be shown based on the complexity of decision tasks for such first-order normal logic programs (Schlipf 1995).

4.3.3 Formalization of the Simplified Ontology

The formalization of the simplified ontology in this work does not cover the generalized ontology. It is relatively easy to extend our representation to cover most of it or the one in (Felfernig et al. 1999). However, some aspects cannot be captured in a straightforward manner in the present form of **DRWCRL**. The lack of arithmetic makes encoding numeric attributes and the constraints on them difficult. For discrete domains, this is in principle possible by introducing an object constant for each value and representing a constraint on them as a set of allowed values. However, this is not very convenient.

Furthermore, a formal visual language for representing configuration knowledge based on the ontology and in a form understandable to domain experts should be developed. The mapping from a configuration model represented in that language to **DRWCRL** should be formalized to allow rigorous analysis of the different variants of configuration tasks and conceptualizations. Such a language and its mapping are needed to implement the approach in this work. Another related research direction would be to formalize the ontology by giving the formal language based on the ontology a direct declarative semantics instead of mapping it to another language. Such a semantics should incorporate some form of the groundedness condition.

5 Conclusions

Many companies design and manufacture configurable products, i.e. products that are routinely adapted to satisfy the specific needs of customers. For such products, the configuration task can be very complex due to the large number of components in the product and the large number of or highly complex interdependencies of the rules of adaptation. Therefore, product configurators, information systems that configure a product or support a person in doing it, are needed.

Numerous diverging theoretical models of configuration tasks and product configurators have been developed within the field of artificial intelligence. The reports on these models have usually addressed the problems of developing a conceptualization and language for representing the configuration knowledge and devising a problem solving method for supporting configuration tasks. In addition, implementations of the languages and methods as well as empirical evidence of their suitability to real-world configuration problems have been reported.

Despite the research no widely accepted theoretical model of configuration knowledge and tasks that would cover all the relevant aspects in a satisfactory manner has emerged. Most of the models presented also lack a sound formal basis that would allow a rigorous analysis and comparison of the models and their implementations. Only few models combine a detailed conceptualization with its formalization. Thorough empirical study of the efficiency and applicability of the implemented algorithms and systems is also mostly missing. The empirical study is made difficult by the confidentiality of knowledge on real products and the huge effort involved in modeling complex products.

This thesis attempts to provide a more unified and formal treatment of configuration knowledge and tasks. A generalized ontology of product configuration that combines the major conceptual approaches to configuration knowledge representation was defined. In addition, declarative languages for representing configuration knowledge were developed. Furthermore, a simplification of the ontology was formalized using the most expressive of these languages. Separately defining a conceptualization, formal languages and a formalization worked well in the situation where it was not clear what exactly the conceptualization would be. This approach facilitates trying out different variants of conceptualizations with relatively little formalization effort.

The languages and the formalization in this work are novel in that their declarative semantics includes the notion of groundedness which means that a configuration must not contain anything that is not mentioned in the configuration model and needed for the product to work. Incorporating this property and other configuration specific constructs in the languages made the formalization of configuration knowledge straightforward. It also allows a uniform representation of the different concepts.

The computational complexity of the configuration tasks for the languages and relative expressiveness of the languages were analyzed. For most of the languages and the configuration task based on the simplified ontology, the task is **NP**-complete. It could be shown that the groundedness property increases in a clearly definable manner the expressive power of a language.

Most previous approaches have not precisely classified the complexity of the configuration tasks. In this respect, this work provides new information on the complexity of configuration tasks and shows that rather expressive combinations of constructs can be used without increasing the complexity beyond **NP**, given certain relatively natural assumptions.

Preliminary empirical evidence for the suitability of the ontology for modeling real products and for the feasibility of implementing configuration tasks on the basis of the formalization was found by modeling simple example products. It may be that the in the worst case intractable configuration problem, as implied by the complexity result, is not quite so hard in practice. This may be due to the structured nature of products designed by humans. Further work in providing empirical evidence for the approach and to validate the structured nature is needed, though.

The main conclusion from this work is that there is no generally accepted formal model unifying the different approaches to product configuration. There is no consensus on the definition of configuration task and configuration ontology. This is witnessed by the abundance of different approaches. However, the different approaches are slowly coming together. Some of them have also become more or less generally accepted cornerstones. The results in this work show that the different approaches can be brought together under a formal framework. It is also clear that more theoretical work on developing formal models and analyzing them is needed. Further theoretical work is also needed for extending the approach in this work to better suit real configuration problems.

References

Artale A., Franconi E., Guarino N. and Pazzi L. Part-whole relations in object-centered systems: An overview. *Data & Knowledge Engineering*(20):347-83. 1996.

Axling T. and Haridi S. A tool for developing interactive configuration applications. *Journal of Logic Programming,* 19:658-79. 1994.

Baader F., Bürckert H.-J., Günter A. and Nutt W. *Proceedings of the Workshop on Knowledge Representation and Configuration WRKP'96.* DFKI GmbH (German Research Center for Artificial Intelligence). Document D-96-04. 1996.

Baader F. and Hollunder B. Computing Extensions of Terminological Default Theories. In Lakemeyer, G. and Nebel, B. (eds.) *Foundations of Knowledge Representation and Reasoning*, Springer-Verlag. Pages 30-52. 1994.

Balkany A., Birmingham W.P. and Tommelein I.D. An analysis of several configuration design systems. *AI EDAM,* 7(1):1-17. 1993.

Buchheit M., Klein R. and Nutt W. Constructive Problem Solving: A Model Construction Approach towards Configuration. TM-95-01. Deutsches Forschungszentrum für Künstliche Intelligenz GMBH (DFKI). 1995.

Bürckert H.-J., Nutt W. and Seel C. The Role of Formal Knowledge Representation in Configuration. In Baader, F., Bürckert, H.-J., Günter, A., and Nutt, W. (eds.) *Proceedings of the Workshop on Knowledge Representation and Configuration WRKP'96*, DFKI GmbH (German Research Center for Artificial Intelligence). 1996.

Carson C. Intelligent Sales Configuration. *PC AI.* 1997.

Clarke B.R. Knowledge-based configuration of industrial automation systems. *International Journal of Computer Integrated Manufacturing,* 2(6):346-55. 1989.

Cohn A.G. Taxonomic reasoning with many-sorted logics. *Artificial Intelligence Review,* 3:89-128. 1989.

Cunis R., Günter A., Syska I., Peters H. and Bode H. PLAKON — An Approach to Domain-Independent Construction. *The Second International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems IEA/AIE-89*, pages 866-74. 1989.

Darr T., McGuinness D. and Klein M. Special Issue on Configuration Design. *AI EDAM,* 12(4). 1998.

Dix J. Semantics of Logic Programs: Their Intuitions and Formal Properties. In Fuhrmann, A. and Rott, H. (eds.) *Logic, Action and Information -- Essays on Logic in Philosophy and Artificial Intelligence*, DeGruyter. Pages 241-327. 1995.

Faltings B. and Freuder E.C. *Configuration—Papers From the 1996 AAAI Fall Symposium.* Technical report FS-96-03. Menlo Park, California: AAAI Press. Technical report FS-96-03. 1996.

Faltings B. and Freuder E.C. Special Issue on Configuration. *IEEE Intelligent Systems & Their Applications*:29-85. 1998.

Faltings B., Freuder E.C., Friedrich G.E. and Felfernig A. Configuration Papers from the AAAI Workshop. AAAI Press. 1999.

Felfernig A., Friedrich G.E. and Jannach D. UML As Domain Specific Language for the Construction of Knowledge Based Configuration Systems. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering SEKE99.* 1999.

Fleischandelr G., Friedrich G.E., Haselböck A., Schreiner H. and Stumptner M. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems & Their Applications,* 13(4):59-68. 1998.

Fowler M. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley. 1997.

Friedrich G.E. and Stumptner M. Consistency-Based Configuration. In Faltings, B., Freuder, E.C., Friedrich, G.E., and Felfernig, A. (eds.) *Configuration Papers From the AAAI Workshop*, AAAI Press. Pages 35-40. 1999.

Gelle E. *On the generation of locally consistent solution spaces.* Ph.D. Thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland. 1998.

Ginsberg M.L. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann Publishers. 1987.

Gruber T. Ontolingua: A Mechanism to Support Portable Ontologies. Version 3.0. KSL 91-66. Stanford University, Knowledge Systems Laboratory. 1992.

Gruber T. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition,* 5:199-220. 1993.

Gruber T. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies,* 43:907-28. 1995.

Gruber T., Olsen G.R and Runkel J.T. The configuration design ontologies and the VT elevator domain theory. *International Journal of Human-Computer Studies,* 44:569-98. 1996.

Guarino N. Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human-Computer Studies,* 43:625-40. 1995.

Guarino N. Understanding, building and using ontologies. *International Journal of Human-Computer Studies,* 46:293-310. 1997.

Guarino N. and Giaretta P. Ontologies and Knowledge Bases Towards a Terminological Clarification. In Mars, N.J.I. (ed.) *Towards Very Large Knowledge Bases.* Amsterdam, IOS Press. 1995.

Günter A., Cunis R. and Syska I. Separating Control From Structural Knowledge in Construction Expert Systems. *Proc. of the Third International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 90)*, ACM, pages 601-10. 1990.

Hales H.L., ed. Automating and Integrating the Sales Function: How to Profit From Complexity and Customization. *Enterprise Integration Strategies,* 9(11):1-9. 1992.

Hodges W. Elementary Predicate Logic. In Gabbay, D. and Guenthner, F. (eds.) *Handbook of Philosophical Logic*, D. Reidel Publishing Company. Pages 1-131. 1983.

Jüngst W. and Heinrich M. Using resource balancing to configure modular systems. *IEEE Intelligent Systems & Their Applications,* 13(4):50-8. 1998.

Klein R. A Logic-Based Description of Configuration: the Constructive Problem Solving Approach. In Faltings, B. and Freuder, E.C. (eds.) *Configuration—Papers From the 1996 AAAI Fall Symposium*, AAAI Press. Pages 1-10. 1996.

Klein R., Buchheit M. and Nutt W. Configuration As Model Construction: The Constructive Problem Solving Approach. In *Proceedings of Artificial Intelligence in Design 1994.* Pages 201-18. 1994.

Knuth D.E. *The Stanford GraphBase A Platform for Combinatorial Computing.* ACM Press. 1994.

Kramer B.M. Knowledge-Based Configuration of Computer Systems Using Hierarchical Partial Choice. *Proc. of the Third International Conference on Tools for Artificial Intelligence TAI 91*, IEEE, pages 368-75. 1991.

Lakemeyer G. and Nebel B. Foundations of Knowledge Representation and Reasoning. A Guide to This Volume. In Lakemeyer, G. and Nebel, B. (eds.) *Foundations of Knowledge Representation and Reasoning*, Springer-Verlag. Pages 1-12. 1994.

Lloyd J.W. *Foundations of Logic Programming.* Second ed. Springer-Verlag. 1987.

Mackworth A.K. Consistency in Networks of Relations. *Artificial Intelligence,* 8:99-118. 1977.

McDermott J. R1: a rule-based configurer of computer systems. *Artificial Intelligence,* 19(1). 1982.

McGuinness D. and Wright J.R. An Industrial-strength Description Logic-Based Configurator Platform. *IEEE Intelligent Systems & Their Applications,* 13(4):69-77. 1998.

McGuinness D. and Wright J.R. Conceptual modelling for configuration: A description logic-based approach. *AI EDAM,* 12(4):333-44. 1998.

Mittal S. and Falkenhainer B. Dynamic Constraint Satisfaction Problems. *Proc. of the 8th National Conference on Artificial Intelligence (AAAI-90)*, MIT Press, pages 25-32. 1990.

Mittal S. and Frayman F. Towards a Generic Model of Configuration Tasks. *Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1395-401. 1989.

Najman O. and Stein B. A Theoretical Framework for Configuration. In Belli, F. and Radermacher, F.J. (eds.) *Proceedings of Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, 5th International Conference, IEA/AIE 92*, Springer-Verlag. Pages 441-50. 1992.

Owsnicki-Klewe B. Configuration As a Consistency Maintenance Task. *Proc. of Künstliche Intelligenz, GWAI-88*, Eringerfeld, Springer-Verlag, pages 77-87. 1988.

Papadimitriou C.H. *Computational Complexity*. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc. 1994.

Peltonen H., Männistö T., Alho K. and Sulonen R. Product Configurations—An Application for Prototype Object Approach. *Proc. of the 8th European Conference on Object-Oriented Programming (ECOOP '94)*, Springer-Verlag, pages 513-34. 1994.

Przymusinska H. and Przymusinski T. Semantic Issues in Deductive Databases and Logic Programs. In Banerji, R.B. (ed.) *Formal Techniques in Artificial Intelligence A Sourcebook*, Elsevier Science Publishers B.V. (North-Holland). Pages 321-67. 1990.

Richardson T. *Using Information Technology During the Sales Visit.* Cambridge, UK: Hewson Group. 1997.

Sabin D. and Freuder E.C. Configuration As Composite Constraint Satisfaction. In Faltings, B. and Freuder, E.C. (eds.) *Configuration—Papers From the 1996 AAAI Fall Symposium*, AAAI Press. Pages 28-36. 1996.

Sabin D. and Weigel R. Product configuration Frameworks—a survey. *IEEE Intelligent Systems & Their Applications,* 13(4):42-9. 1998.

Schlipf J.S. Complexity and undecidability results for logic programming. *Annals of Mathematics and Artificial Intelligence,* 15:257-88. 1995.

Schreiber A.T. and Birmingham W.P. The Sisyphus-VT initiative. *International Journal of Human-Computer Studies,* 44(3). 1996.

Schreiber G. and Wielinga B. Configuration-Design Problem Solving. *IEEE Expert*:49-56. 1997.

Schröder C., Möller R. and Lutz C. A Partial Logical Reconstruction of PLAKON/KONWERK. In Baader, F., Bürckert, H.-J., Günter, A., and Nutt, W. (eds.) *Proceedings of the Workshop on Knowledge Representation and Configuration WRKP'96*, DFKI GmbH (German Research Center for Artificial Intelligence). 1996.

Searls D.B. and Norton L.M. Logic-based configuration with a semantic network. *Journal of Logic Programming*:53-73. 1990.

Simons P. *Extending and Implementing the Stable Model Semantics*. Ph.D. Thesis, Helsinki University of Technology, Finland. 2000.

Soininen T. *Product configuration knowledge: Case study and general model.* Master's thesis, Helsinki University of Technology. 1996.

Soininen T. and Niemelä I. Formalizing Configuration Knowledge Using Rules with Choices. Technical report TKO-B142. Laboratory of Information Processing Science, Helsinki University of Technology. 1998.

Stumptner M., Friedrich G.E. and Haselböck A. Generative constraint-based configuration of large technical systems. *AI EDAM,* 12(4):307-20. 1998.

Stumptner M. and Haselböck A. A Generative Constraint Formalism for Configuration Problems. *Advances in Artificial Intelligence*, Berlin, Springer-Verlag Berlin Heidelberg, pages 302-13. 1993.

Syrjänen T. *A Rule-Based Formal Model for Software Configuration.* Master's thesis, Helsinki University of Technology, Department of Computer Science, Laboratory for Theoretical Computer Science. 1999.

Tiihonen J. *Computer-assisted elevator configuration.* Master's thesis, Helsinki University of Technology. 1994.

Tiihonen J. and Soininen T. Product Configurators – Information System Support for Configurable Product. Technical Report TKO-B137. Laboratory of Information Processing Science, Helsinki University of Technology. 1997.

Tiihonen J., Soininen T., Männistö T. and Sulonen R. State-of-the-Practice in Product Configuration—A Survey of 10 Cases in the Finnish Industry. In Tomiyama, T., Mäntylä, M., and Finger, S. (eds.) *Knowledge Intensive CAD.* London, Chapman & Hall. Pages 95-114. 1996.

Tiihonen J., Soininen T., Männistö T. and Sulonen R. Configurable Products - Lessons Learned From the Finnish Industry. In *Proceedings of 2nd International Conference on Engineering Design and Automation*, Integrated Technology Systems, Inc. 1998.

Tong C. and Sriram D. Introduction. In *Artificial Intelligence in Engineering Design*, Academic Press Inc. Pages 1-53. 1992.

Weida R. Closed Terminologies in Description Logics. In Faltings, B. and Freuder, E.C. (eds.) *Configuration—Papers From the 1996 AAAI Fall Symposium*, AAAI Press. Pages 11-18. 1996.

Weigel R. and Faltings B. Abstraction Techniques for Configuration Systems. In Faltings, B. and Freuder, E.C. (eds.) *Configuration—Papers From the 1996 AAAI Fall Symposium*, AAAI Press. Pages 55-60. 1996.

Yu B. and Skovgaard H.J. A configuration tool to increase product competitiveness. *IEEE Intelligent Systems & Their Applications,* 13(4):34-1. 1998.