Mag. Anna Ryabokon

KNOWLEDGE-BASED (RE)CONFIGURATION OF COMPLEX PRODUCTS AND SERVICES

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

Alpen-Adria-Universität Klagenfurt
Fakultät für Technische Wissenschaften

Betreuer und Erstgutachter
O.Univ.-Prof. Dipl.-Ing. Dr. Gerhard Friedrich
Alpen-Adria-Universität Klagenfurt, Institut für Angewandte Informatik

Zweitgutachter
Univ.-Prof. Dipl.-Ing. Dr. Axel Polleres
Wirtschaftsuniversität Wien, Institut für Informationswirtschaft

Klagenfurt, 02/2015

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich

- die eingereichte wissenschaftliche Arbeit selbständig verfasst und andere als die angegebenen Hilfsmittel nicht benutzt habe;
- die während des Arbeitsvorganges von dritter Seite erfahrene Unterstützung, einschließlich signifikanter Betreuungshinweise, vollständig offengelegt habe;
- die Inhalte, die ich aus Werken Dritter oder eigenen Werken wortwörtlich oder sinngemäß übernommen habe, in geeigneter Form gekennzeichnet und den Ursprung der Information durch möglichst exakte Quellenangaben (z.B. in Fußnoten) ersichtlich gemacht habe;
- die Arbeit bisher weder im Inland noch im Ausland einer Prüfungsbehörde vorgelegt habe und dass
- die zur Plagiatskontrolle eingereichte digitale Version der Arbeit mit der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine tatsachenwidrige Erklärung rechtliche Folgen haben wird.

Unterschrift: _____

Klagenfurt, 28. Februar 2015

Zusammenfassung

Die *Produktkonfiguration* ist eine erfolgreiche Anwendung der Künstlichen Intelligenz. Aus einem Katalog verfügbarer Komponenten werden auf Basis spezifischer Anforderungen Konfigurationen ermittelt. Beispiele für Konfigurationsprobleme sind Telefonanlagen, elektronische Eisenbahnstellwerke oder Automatisierungssysteme. Oft beanspruchen solche Probleme eine komplexe Optimierung, welche die Lösungen mit einer minimalen Anzahl von Komponenten bevorzugt, um so die Herstellungskosten zu minimieren.

Das *Rekonfigurationsproblem* existiert dann, wenn ein konfiguriertes Produkt oder Service umgestaltet werden muss. Das ist eine wichtige Aktivität im Kundendienstzyklus (After-Sales-Bereich) der Firmen, die konfigurierbare Produkte verkaufen. Typischerweise solche Produkte haben die lange Lebensdauer und die Konfiguration- sowie Benutzeranforderungen ändern sich gleichgehend.

Die Entwicklung von wissensbasierten Systemen benötigt eine Repräsentationssprache, die einerseits ausdrucksstark genug ist um das Konfigurations- bzw. Rekonfigurationsmodel zu erfassen und die andererseits die Anwendung effizienter Schlussfolgerungsmethoden erlaubt. Der Schwerpunkt dieser Dissertation liegt auf der Entwicklung von *wissensbasierten Darstellungssprachen* für die unterschiedlichen (Re-)Konfigurationsprobleme von Siemens und im Allgemeinen. Die Beweisführung wurde mittels Übersetzung der Wissensbasis in aktuelle und effiziente Formalismen wie zum Beispiel Answer Set Programming und Constraint Programming erbracht.

In der Praxis kann die Ermittlung einer Lösung für (Re-)Konfiguration Probleme aufgrund der großen Komponentenanzahl und der Präsenz symmetrischer Lösungen scheitern. Deswegen wurde eine *Dekompositionsmethode* entwickelt, welche die Aufteilung dieser Probleme in Sub-Probleme zulässt. Für jedes dieser Sub-Probleme kann dann leichter eine Lösung ermittelt werden. Unter Wahrung der allen Lösungen, die Methode verbessert effektiv die Ausführungszeit und Qualität den Lösungen für real existierende Konfiguration Probleme.

Abstract

Product configuration is a successful AI application which operates by selecting and assigning components from a catalog in accordance with the customer and configuration requirements. Configuration problems occur frequently during configuration of telephone switching systems, electronic railway interlocking systems, automation systems, etc. Often such problems are subject to complex optimization preferring solutions which include a minimal number of components, thus, minimizing the overall production costs.

A *reconfiguration* problem arises when a product or service is not designed from scratch, but has parts of the existing configuration adapted. This is an important activity in the after-sale lifecycle for companies selling configurable products or services. Typically such products have a long lifetime and their configuration requirements are changing in parallel with the customers' business.

The development of knowledge-based (re)configuration systems requires the application of a knowledge representation language which on the one hand is expressive enough to capture a (re)configuration model, and on the other, there should exist reasoning methods for it that allow a solution be computed efficiently. This PhD project focuses on the development of a *knowledge representation language* which allows encoding of different (re)configuration problems occurring in practice of Siemens and in general. The reasoning is done by translating the knowledge base to modern formalisms such as Answer Set Programming and Constraint Programming.

In practice finding a solution for (re)configuration problems using general frameworks might result in an unacceptable performance, because of the large number of components and the presence of symmetrical solutions. Therefore, a *decomposition method* was developed that allows partitioning of these problems into loosely coupled sub-problems for which solutions can be computed more easily than a solution to the whole problem. While preserving all solutions, the method significantly improves performance and quality of solutions for real world configuration problems.

Acknowledgments

This work was supported by the Austrian research and promotion agency FFG as part of the Reconcile project (grant number 825071). It has been carried by the Intelligent Systems and Business Informatics group of the Applied Informatics Institute at the Alpen-Adria Universität Klagenfurt. In the process of developing this thesis many people have helped me to shape my thoughts. I would, however, like to highlight a few whose support was especially valuable. Without all of you this work would not have been possible!

First of all, I am grateful to Prof. Gerhard Friedrich who gave me the opportunity to work in his research group and in the Reconcile project, and who offered constructive scientific assistance and valuable criticism as my adviser. My thanks also go to Prof. Axel Polleres for his scientific advice and helpful suggestions.

I am indebted to all Siemens colleagues of the Reconcile project: Dr. Andreas Falkner, Dr. Alois Haselböck, Dipl.-Ing. Gottfried Schenner and Dipl.-Ing. zSPM Herwig Schreiner, who have shared their experience with me. My thanks to all of you for your numerous comments and questions, engaging discussions during our meetings and excellent attitude towards teamwork.

I would like to acknowledge the participants of the Constraint Satisfaction for Configuration project for collaboration and thought-provoking discussions. I am particularly grateful to Prof. Georg Gottlob for his insightful comments regarding my presentations which deepened my enthusiasm for research work.

My sincere thanks also go to Mag. Wauki Hall and Anthony Hall, MA, for proof-reading of some parts of my thesis and for improving my English skills. Because of their professional competence in teaching and enormous friendliness, I really enjoyed our discussions.

And last but not least, I warmly want to thank my family, chiefly my husband and my children for their love, instructive optimism and patience over the last four years.

Contents

List of Figures

List of Tables

CHAPTER

1 Introduction

Product configuration is one of the successful applications of Artificial Intelligence (AI) which has attracted many researchers over the last three decades. A configuration problem corresponds to a complex design activity, which can be defined as follows: given a catalog of components, compose an artifact such that the complex requirements reflecting the individual needs of a customer and the compatibility of a system's structures are satisfied. The configuration of products and services is a fully or partially automated process which is often implemented by a knowledge-based information system. The challenges, theoretical advances and latest research results of a configuration problem are actively discussed by both academia as well as industrial partners at International Configuration Workshops, major AI conferences such as IJCAI, AAAI, ECAI and in special issues on configuration in journals such as AI Communications, IEEE Intelligent Systems or AIEDAM.

Configuration

Configuration problems occurring in different domains have different characteristics. The most challenging problems originate in large technical domains such as railway interlock systems, telephone switches, electric power distribution systems, etc. In extreme cases, instances of these problems comprise hundreds or even thousands of interconnected components that are subject to multiple requirements. Finding valid solutions, called configurations, for such systems is nearly impossible for human beings, and pushes modern configuration systems to their limits. Smaller but more frequently appearing configuration problems can be found in the field of mass customization. Such configuration systems allow the personalized design of cars, computers, hotel accommodation services, etc. Even the assignment of papers to reviewers can be implemented by a configuration system. Despite

being smaller than technical ones, the mass customization problems are by no means simpler. The reason is that these systems often require an online interactivity between a configurator and a customer. The latter specifies the desired properties of a configurable product and the configuration system has to compute a product which satisfies them very fast. Therefore, the development of such configurators requires the design of efficient algorithms able to calculate a (partial) configuration within a limited time.

In addition, regardless of the application domain, a configuration system might be required to find some preferred solution among all existing ones. For instance, in the domain of technical configuration there is often a necessity for the number of software modules to be minimized in order to reduce the complexity of a system's architecture. This might also be the case for configurators developed for use in the domain of mass customization. In such scenarios the customers might need to design a computer with certain predefined features and/or minimal costs; to put a dog food together which maximizes the content of vitamin A and minimizes the fat content. The incorporation of the mentioned (multiple) objectives, also called preferences, implies the equipping of knowledge-based configurators with optimization methods able to deliver solutions in a time which is reasonable for the application domain.

Reconfiguration

Reconfiguration is another important and difficult task in the after-sale life-cycle of configurable products and services. Quite often requirements for these products and services are not stable and change over time. In this case, a previously consistent configuration has to be adapted in order to reflect changes that have emerged. To keep a product or a service up-to-date a re-engineering organization (system) has to decide which modifications should be introduced to the existing configuration, such that the new requirements are satisfied. The generation of the modified configuration, i.e. reconfiguration, is for several reasons a complex optimization process. First, the reconfiguration process must decide which changes must be introduced to a legacy solution. This task is usually split into two sub-tasks: (i) find which parts of a legacy configuration are reused in a new configuration, and (ii) extend the reused parts with a set of interconnected components. The artifact resulting from these changes has to satisfy all the modified customer and system requirements. Second, the costs of changing a legacy product or service must be minimized. This requirement in particular is important for the modification of existing products, since the introduction of changes to existing

physical products might be extremely costly or even impossible.

Contributions and thesis structure

Despite its extreme importance, reconfiguration tasks have only been studied superficially in the literature. In this work we close this gap and develop a unified approach that can handle both configuration and reconfiguration problems. Our main contribution is the development of a knowledge representation language which is expressive enough to represent a variety of (re)configuration problems occurring in practice. The language allows (re)configuration knowledge such as different customer and system requirements, and/or information describing the legacy configuration and possible changes to it to be encoded. Moreover, the language supports definition of (re)configuration costs and preference criteria.

In particular, we make the following contributions:

(1) During our research into product configuration we study typical common and generic characteristics of the (re)configuration problems. Most of the representative problem instances considered in the work correspond to real-world application cases of our industrial partner Siemens. However, some other configuration tasks which might occur in practice are also considered.

(2) In order to develop the knowledge-representation language for (re)configuration of complex products and services we analyze existing automated configuration systems, called configurators, as well as the knowledge representation and reasoning (KRR) approaches used in them. Different formalisms are concerned such as rule-based, case-based and model-based approaches. Among these are model-based approaches such as Boolean satisfiability (SAT) and constraint satisfaction problem (CSP) solving, which then are studied in greater depth because: (a) their usability is well-established; (b) they perfectly match the nature of the configuration process and (c) they are efficiently used in modern configuration systems.

(3) On this basis we elaborate a formal method for defining (re)configuration problems using a logic-based knowledge representation language. The language can be used to model any of the representative test cases mentioned above. This means that the proposed formalism allows a variety of (re)configuration requirements as well as domain-specific costs capturing customer and/or system preferences to be represented. These preferences are important in practice due to their optimization. For instance, the inclusion of each component in a configuration is usually associated with additional

costs, therefore, minimization of the number of components used results in reduction of overall costs of configurable products or services.

(4) Encodings of the problem instances given in our language are easily mapped to Answer Set Programming (ASP) or Constraint Programming (CP) representations. The latter can then be solved by any of the publicly available ASP or CP solvers. In order to improve the performance of state-of-the-art solvers for configuration and reconfiguration problems we design and implement various heuristics, solving strategies (parameter tuning) and symmetry breaking techniques.

(5) To prove their feasibility the implementations are evaluated on the set of real-world configuration and reconfiguration problem instances. The experimental results show that the suggested methods can be applied in practice since they demonstrate an admissible efficiency. However, for some problem instances a number of performance issues are identified due to the large number of available components and/or complex relations between them. This is particularly the case for the problem instances where a customer specifies multiple preference criteria for a desired (re)configuration.

(6) Consequently, specific algorithms are required to overcome the computational hardness of the problems. Thus, we suggest a decomposition method which produces a considerable improvement in the quality of the solution found. We prove that our method preserves all solutions and significantly accelerates the search process. Moreover, applied to a set of industrial problems the method results in up to 525% better solutions with respect to the specified preference criteria.

The remainder of this thesis is organized as follows.

Chapter 2 starts with a short historical overview of knowledge-based configuration. Then we discuss the configuration problem and analyze existing knowledge representation and reasoning approaches. For each approach we provide examples of how configuration knowledge can be represented using these approaches and outline their pros and cons. In the second part of the chapter we address the reconfiguration problem and highlight the state-of-the-art reconfiguration approaches.

We introduce our knowledge representation language for (re)configuration problems in Chapter 3. We begin with an introductory example of a configuration problem that demonstrates particular features of the problem. Next, we provide formal definitions of configuration and reconfiguration problems as well as their optimization variants. Further, we present a logic-based formalism which allows a unified knowledge representation and solution of both configuration and reconfiguration problems. Finally, a translation of the developed language to Answer Set Programming as well as some modeling patterns are presented.

Chapter 4 describes three real-world application cases: the Partner Units Problem, the House problem and the Reviewer Assignment Problem. Each of these problems can be formalized in our knowledge representation language for both configuration and reconfiguration scenarios. Additionally, we demonstrate the translation of encodings of (re)configuration problems to ASP. An evaluation of the developed formalism using ASP solvers is performed for each problem. For the Partner Units problem and the House problem we also present the evaluation results obtained using CP solvers. Moreover, we introduce and evaluate some advanced solving methods such as heuristic search, symmetry breaking strategies and efficient solver parametrization techniques. At the end of the chapter we analyze the experimental results by comparing them to the results achieved recently by other researchers.

In Chapter 5 we suggest a novel approach for rewriting the tuple generating dependencies used in our knowledge representation language for expression the existence of certain configuration components. We further prove that the suggested algorithms are sound and complete, and present implementation details. To show the feasibility of the method we evaluate our approach and discuss experiments which show the effectiveness of the method in terms of runtime and quality of computed solutions for a set of industrial benchmarks.

Finally, in the last chapter we provide a summary of our contributions, outline the overall results and present open problems for future research.

Impact

Parts of the work presented herein have been published as peer-reviewed articles in the proceedings of several international workshops and conferences, which we now introduce in chronological order.

- We published our knowledge representation language for (re)configuration problems, which is introduced in Chapter 3, in the proceedings of the "IJCAI 2011 Workshop on Configuration" [FRF+11b] and the "International Workshop on Logics for Component Configuration 2011" [FRF+11a]. The evaluation results for the House (re)configuration problem presented in these papers are discussed in Chapter 4.

- We submitted the Partner Units Problem benchmarks to the Third Answer Set Programming Competition 2011. The application of our language to the Partner Units configuration problem using ASP and the experimental results using different general-purpose KRR systems

such as CP, ASP, SAT, etc. were described in the paper presented at the "International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimization problems" [ADF+11]. Some selected parts of this paper are provided in Chapter 4.

- Additionally, we showed the feasibility of our approach for the Reviewer Assignment Problem, when Linked Data on the Web can be used as "component catalog" by distribution of submitted papers among PC members, in the paper appeared in the proceedings of the "International Conference on Web Reasoning and Rule System" [RPF+12]. We recall the implementation details and evaluation results published there in Chapter 4.

- The study about symmetry breaking for ASP-based approaches evaluated on the House problem was published as a technical report [Rya12]. In [FSFR12] we show how the ASP symmetry breaking techniques can be applied in practice during testing object-oriented configurators. The latter publication was presented at the "ECAI 2012 Workshop on Configuration".

- The conflict-based program rewriting approach for solving configuration problems [RFF13], explained in Chapter 5, was presented at the "International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013)".

Besides the mentioned above papers, we have also been involved in publishing a book chapter which provides an overview of different configuration knowledge representation and reasoning approaches for configuration tasks [HFS+14]. This publication discusses major advantages and disadvantages of the most famous KRRs applied to configuration in the last 30 years.

As previously mentioned, the baseline knowledge representation and reasoning approaches discussed in this work have been used to model and solve real-world configuration scenarios [SFRF13] from S'UPREME based configurators. S'UPREME is a configuration engine of Siemens AG which is applied to configure complex large-scaled technical systems such as railway safety systems or industrial production systems within Siemens and there exist more than 30 different applications based on this system [HS14]. The results presented in the thesis can be used as a guideline for the further development of such configurators.

CHAPTER

2 Knowledge-based configuration

2.1 Motivation

According to a simple definition, which can be found in The American Heritage Dictionary of the English Language[1], *configuration* is an arrangement of parts or elements. A similar definition can be found in The British Macmillan Dictionary[2], *configuration* (in computing) is the way in which the different parts of something are arranged. Which parts or elements of a system have to be included in configuration is predefined by a set of complex requirements reflecting individual needs of a customer and compatibility of product structures. Configuration is a fully or partly automated approach supported by a knowledge-based information system called *product configurator*.

Originally configurators appeared because configurable products and services were very complex to be assembled by humans without any support. In many cases these expensive products were produced by order and single-copy, thus requiring a significant number of highly qualified workers developing each of them. Emerging research on expert systems in the 1980s resulted in a number of approaches to knowledge-based configuration, also referred to as product configuration. Historically, the first industrial configurator was introduced in 1982 when McDermott presented R1/XCON 'configurer'. The system was used to configure Digital Equipment Corporation's VAX computer systems [McD82]. The knowledge base of the later versions of this configurator included more than 10000 rules and about 30000 of components capturing all possible aspects of VAX computers. Given a customer's order R1/XCON had to determine whether the order is complete. In case of an incomplete order the system determined an appropriate set of missing components and

[1] www.thefreedictionary.com/configuration

[2] http://www.macmillandictionary.com/dictionary/british/configuration

extended the order. The developers also reported about their intention to include an interactive salesperson's assistance to simplify the extension of the order by involving the customer. Next the configurator computed all justifiable spatial relationships among the required components. The diagrams presenting these relationships were used by the technicians who assembled the ordered computer. R1/XCON supported screening of an order during the manufacturing process. This feature prevented the customer from 'unconfigurable' choices.

Since then results of the AI research have been adopted by such companies as ConfigIt, ILOG, Oracle, SAP, Siemens, Tacton (provided in alphabetical order) which maintain leading industrial systems in this area. The development of an efficient configurator allows to assemble large complex technical systems and might reduce the total production costs significantly.

Researchers in both academia and industry have been mostly focused on the development of general purpose configurators which would be applicable to any specific domain. Several prototypes were created in technical domains. For example, PLAKON [CGS+89] and KONWERK [Gün95] are systems applicable to different domains such as configuration of passenger cabins in vehicle aircrafts, configuration of hydro-geological models and elevators, modeling of bills of materials in automobile production, arrangement and dimensioning of drive control systems, etc. PROSE [WWV+93b] is a general configuration platform for telecommunication hardware which supports sales and order processing at AT&T Network Systems. An effective application of COCOS configuration tool in LAVA configurator developed for telephone switching systems that consisted of about 1000 frames, 30000 modules, 10000 cables and 2000 other units is described in [SHF94, FFH+98]. The practice showed that the application of LAVA was able to reduce up to 60% of the configuration costs which constitute approximately 20% of the total product costs.

With the lapse of time the focus has been shifted more in the direction of *mass customization* allowing to design individual products desired by a customer at a cost similar to mass production [SBF01]. Currently configurators cover a wide range of customers and can be found in practically every price segment. You can configure your own car, bicycle, computer, skis and even a forage for a dog. Web plays a key role in progress of configuration systems/tools providing an essential interactive connection between customer and producer. This imposes restriction on time of search for a configuration, but gives a possibility to buy products online. Interestingly, the number of configurators is rapidly growing with the presence of the Web. Tiihonen et al. [THAS13] reported that

in 2009 the number of web-based configurators listed in the International Configurator Database[3] was equal to 580 (retrieved on 27.11.2009). In 2013 the database lists already 824 configurators (retrieved on 19.11.2013) which is about 42 % of growth in 4 years. Web-based configuration is a wide field of research which is beyond a scope of this work and will be superficially discussed. For a detailed analysis about product configurators listed in the International Configurator Database see the technical report released by Austrian media specialist cyLEDGE Media [BPS13], which includes market study, evaluation of different criteria, etc.

This work focuses mostly on industrial configuration problems occurring in Siemens practice. These problems correspond to combinatorial problems which include several optimization criteria. They require selection and assignment of hardware modules depending on the customer and configuration (system) requirements and occur frequently during configuration of technical systems produced by Siemens such as telephone switching systems, electronic railway interlocking system, automation systems, etc.

2.2 Configuration problem

A configuration problem corresponds to a composition activity in which a desired configurable product is assembled by relating individual components of predefined types. The components and relations between them are usually a subject to constraints expressing their possible combinations allowed by the system's design. The types of the components, relations between them as well as additional constraints on sets of related components constitute *configuration requirements*. These requirements are defined for a configurable product or service usually once and remain constant during the configurator's life-cycle, unless the subject of a configurator is redesigned. Customizations of products or services requested by a customer are captured by a set of *customer requirements* which can be changed each time the configurator is executed. Moreover, often it is required to find a configuration that is preferred with respect to some criteria, e.g. the cheapest configuration or a configuration including the least number of some specific components, etc. Such preferences of a customer as well as of a service or product provider are expressed by a set of *preference criteria*.

Definition 1 (Configuration problem instance)**.** Let $REQ = CR \cup CT$ be a set of requirements comprising a set CR of configuration requirements and a set CT of customer requirements and

[3] http://www.configurator-database.com/database

OPT be a set of preference criteria. Then, a tuple $CPI = \langle REQ, OPT \rangle$ is a configuration problem instance.

Definition 2 (Configuration)**.** Given a configuration problem instance $\langle REQ, OPT \rangle$, a solution S of the configuration problem instance, called *configuration*, is a set of components and relations between them satisfying the configuration and customer requirements given in REQ.

An *optimal configuration* is such problem solution S that there is no other solution S' that is preferred over S with respect to preference criteria given in OPT.

Computation of a valid solution corresponds to a decision problem, i.e. to find out whether a solution exists or not within a configuration model. Identification of an optimal configuration is a more complex task, because it requires enumeration of multiple solutions in order to find the optimal configuration which maximizes or minimizes the predefined preference criteria.

Similarly to any problem solving activity, *solution of a configuration problem* includes the main steps: a) development of a conceptual model, b) encoding of the model in a selected knowledge representation language and c) identification of a(n optimal) solution using appropriate algorithms. In most of the configuration systems existing today the domain knowledge, captured by configuration and customer requirements, is modeled in terms of component types and relations between them. Each type is characterized by a set of attributes which specify features of the component. These features describe functional and technical properties of real-world and abstract components of a configurable product. An attribute takes values from a predefined domain which can be discrete or continuous. An attribute can take a single value or a set of multiple values representing strings, numbers, types of components, etc. The latter are often referred to as ports in the literature, because they express possible connections between components. Creation of connections (relations) between components is an important activity of the configuration process. Usually each component type can be connected with one or many component types. Possible connections between two component types are usually modeled as relations, where cardinality of a relation expresses the number of components that can be connected to each other. In most of the cases, modeling languages used in configuration allow to specify relations of the following types: classification (is-a), aggregation/composition (part-of), association (customer defined relations). These relations can be defined locally, i.e. between two particular instances, and globally, i.e. for all component types that take part in a relation.

For a successful application of a configuration system it is very important to choose a knowledge

representation language which is expressive enough to capture a configuration model. Usually a configuration model includes a library of component descriptions which are organized in a part-of hierarchy and some additional constraints. In addition, there should exist reasoning methods that allow to compute a solution efficiently.

2.3 Knowledge representation and reasoning for configuration

Since introduction of the first configuration systems researchers have tried different approaches to knowledge representation and reasoning (KRR) including production rules, constraints languages, heuristic search, description logics, and others, see [Stu97, SW98a, SH07, Jun06, FHBT14] for a survey. Although different classifications of configuration systems exist in literature, we consider classification depending on representation language used to describe configuration knowledge because KRR for configuration is the main topic of this work. As suggested by Sabin and Weigel in [SW98a] KRR used in configuration systems can be classified into *rule-based*, *case-based* and *model-based* approaches.

Rule-based approaches are mostly following the ideas of R1/XCON expert system. These configurators use production rule systems such as ILOG JRules[4] or OpenRules[5] for KRR. The original R1/XCON systems used production rules programmed in OPS5 [BFK85]. In this system productions rules are of the form

if *condition* **then** *action*

An OPS5 based system system stores all facts describing the current state of the world in a working memory. The execution of the rules is performed in reorganize/act cycle. If the condition of the rule is satisfied by a state of a working memory, then the system executes the action which modifies the state of the working memory. The rule interpreter acts in a forward chaining manner. At each step the system selects a set of rules whose bodies are satisfied and executes their actions. The process continues until one of the conditions is met: a) there are no rules which bodies are satisfied by a memory state, b) "halt" action is executed or c) the algorithm exceeds a given number of iterations.

On the one hand, the usage of production rules allowed to overcome limitations of traditional procedural programming languages. Namely, identification of conditions required to execute an

[4]http://www-01.ibm.com/software/websphere/products/business-rule-management/
[5]http://openrules.com/

action and composition of a sequence of actions needed to obtain a valid configuration. On the other hand, exploitation of rule-based systems made it clear that the management of large rule bases is problematic [MSM88, SW98a, GK99]. The source of the problem lies in the nature of the rules which are used to express both relations between component types and actions. The relations are a part of domain knowledge describing compatibility of component types, their dependencies, etc., whereas the actions capture procedural knowledge describing a way to obtain a solution. As a result knowledge about a single concept is often spread among different rules. In case of changes to a concept the programmer has to make sure that a set of changed rules covers all aspects of the required change. Identification of this set of rules can be extremely difficult. Attempts to solve the maintenance problem by introduction of meta rules [Bac88] failed for the large rule bases and the problem remains largely unsolved.

Case-based approaches use case-based reasoning (CBR) which exploits knowledge about past solutions, called cases or situations, to solve a new problem. It relies on the idea analogous to the humans' solving process when we remember and reuse or adopt specific problem knowledge of similar situations in a new problem situation. CBR is the repeating and integrated process of solving a problem and learning from this experience. That is, when a problem was solved successfully in the past, the experience is stored and used in the future. In case of failure the reason is identified and learned in order to avoid the same mistake. If there is no case similar to a new situation, the case-based system has to (automatically) adapt the most similar case to the new situation by means of adaptation rules [AP94].

There are very few research works describing case-based configuration available. One of the examples is a configuration solver enhanced by cases in order to interactively specify, configure and extend telecooperation systems [RV96]. The more recent example can be found in [TCC05] where authors suggest an CBR algorithm that generates a bill of material which meets customers and configuration requirements, and effectively reduces the time and cost of design.

One of the main advantages of CBR systems is the ability to derive (adapt) new configurations (cases) from existing ones. The adaptation is often done by means of transformation rules which express valid changes to a configuration. This feature is useful in application scenarios in which the definition of all configurations can hardly be done from scratch. However, the adaptation process is effective only in situations when minor changes to a configuration are required and transformation rules are available. In practice this is the case when only few configurations of a product exist.

Otherwise, the definition of initial configurations as well as transformation rules is a challenge which avoids broad application of case-based configuration systems.

Model-based approaches have drawn the attention of researches because they allow to overcome the development and maintenance problems of the rule-based and case-based systems. Instead of capturing *experience* knowledge describing how to configure a product or a service, as the production rule systems do, model-based systems maintain a library of models which describe the nature of a configurable product. Such models are versatile and can be reused in different instances of a configurator. Moreover, general model-based problem solving engines do not depend on any features of a product model and can be applied to compute configuration independently of an application domain. Such independence results in high robustness of the model-based systems.

Application of model-based methods allows to separate the domain-specific knowledge, describing decomposable elements of a configurable product and their relations, from task-specific knowledge in terms of problem solving engines. Such separation simplifies composition of models of different products for obtaining configurators of complex solutions, thus perfectly matching the nature of the configuration process, namely, composition of new entities out of existing ones.

Knowledge representation languages used for model-based systems include a variety of formalisms such as ordinary differential equations, finite state machines or predicate calculus, and use different inference methods like theorem proving, constraints satisfaction or optimization [Str08]. We consider model-based paradigms in the following subsections in more details since they mostly represent state-of-the-art research in the field of product configuration.

2.3.1 Constraint-based approaches

In modern configuration systems modeling of the configuration and customer requirements as a constraint satisfaction problem (CSP) is de facto a standard. CSP is defined as a set of variables $V = \{v_1, v_2, \ldots, v_n\}$ where each variable is associated with a corresponding domain $D = \{D_1, D_2, \ldots, D_n\}$. For each variable v_i a domain D_i, denoted also $dom(v_i)$, comprises all values the variable can take. A constraint C is a relation defined on a set of variables $V' \subseteq V$, denoted also $vars(C)$, which defines allowed combinations of the value assignments to the variables $vars(C)$. A (possibly partial) function $va : V \mapsto \bigcup_{D_i \in D} D_i$ is called a variable assignment, if for every $v_i \in V$, $va(v_i) \in dom(v_i)$. If a tuple of values assigned by a variable assignment va to $vars(C)$ is contained

in C then va satisfies C. A variable assignment va is a solution of a CSP, if va is total on V and satisfies all constraints of the problem.

In the pioneering work of Mittal and Frayman [MF89] components, their attributes and relations are expressed as variables and the constraints are used to restrict compositions of the components to the valid ones. Hence, CSP matches perfectly with the requirements to knowledge representation language of a configuration system.

However, it was soon recognized that CSP models are unable to capture the dynamic nature of the configuration problem. That is, in many cases the number of components that should be included in a configuration is unknown a priori. Of course, one can include a sufficient number of variables in a model, such that a configuration can be found. The problem with this solution is that problems with a large number of variables are often hard to solve.

Dynamic CSPs suggested by Mittal and Falkenhainer [MF90] extend a common CSP framework with constraints activation. The latter allows to include additional variables in a solution, if an activation condition is satisfied. Another new concept – compatibility constraints – allows to restrict the solution space by defining that if a component is included in a configuration, then a set of other components must be in it. The main problem of the approach is that one has to differentiate between multiple components of the same type explicitly, i.e. constraints must be duplicated for each corresponding variable.

A resource-based configuration views each component as a provider of some sort of resources (functionality) required in a configuration [Stu97]. Consequently, the set of all components in a configuration has to provide all required resources. The resource-based approach [HJ91] is based on a combination of constraints with production rules. The latter are used for functionality-based selection of components from the library. That is, on each iteration the algorithm retrieves a set of resources required in a configuration, but that are not provided by a set of components of the current (partial) configuration. Using some predefined heuristics the method selects a component that provides some of the currently unavailable resources and adds it to the solution. The algorithm continues until the list of required resources becomes empty, i.e. all constraints are satisfied. The approach suggests inclusion of production rules and constraints for different parts of a knowledge base to influence a solving algorithm. The selection heuristics can easily be implemented as variable orderings in CSP.

Generative CSPs introduced by Stumptner et al. [SFH98] differentiate between variables that

stand for components and variables that represent relations and attributes. Variables of the first type defined in the GCSP can be introduced during solving if the number of components in a configuration should be increased. In general, the approach is similar to the one of object-oriented programming, where classes include only a finite set of attributes and relations, but the number of instances of a class is unlimited. Moreover, GCSP allows the definition of resource constraints as equations over aggregates representing resource functions. Hence, GCSP improves on important features of both dynamic CSP and resource-based approach. The language of GCSP was originally defined in a frame-like manner [SFH98], but then was changed to be similar to object-oriented programming languages to reduce the learning and coding efforts [FH13]. For instance, we model a system to be configured containing at least one and up to two transmitters and from one to five receivers that have to be installed on a main board. Each of the installed components can be digital (i.e. digital = true) or analog (i.e. digital = false). In addition, a configuration requirement states that all used transmitters and receivers have to be of the same type. The problem can be represented as GCSP as follows:

class $Transmitter$
    attr $digital$: $boolean$
class $Board$
class $Receiver$
    attr $digital$: $boolean$
assoc $Transmitter.board(1)$ - $Board.slot\_1(1..2)$
assoc $Receiver.board(1)$ - $Board.slot\_2(1..5)$
constraint $Board.validDevice$: $slot\_1.digital = slot\_2.digital$

2.3.2 Description logics

Description logics (DLs) [BHS08, BCM+10] are a family of knowledge representation languages which are specifically designed to represent domain knowledge in a formal and well-structural way. Typically a DL knowledge base comprises two parts: (i) terminology (TBox) – a set of concept and role definitions expressing classes of objects in an application domain and relations between them; (ii) assertions (ABox) – a set of definitions describing individuals occurring in the domain in terms of concepts and roles given in TBox. The basic definitions in a DL knowledge base include atomic concepts (unary predicates), atomic roles (binary predicates) and individuals (constants).

More complex expressions – axioms – are built using Boolean constructors, such as conjunction ($\sqcap$), disjunction ($\sqcup$) and negation ($\neg$), as well as existential ($\exists r.C$) and value ($\forall r.C$) restrictions. The simplest terminological axioms of a DL are descriptions which use subsumption ($\sqsubseteq$) to provide a definition of a concepts. A concept C is subsumed by D if every instance of C is also an instance of D. For instance, the definition of a component type C_1 that is a subtype of C_2 and which is related by relation rel to some component of the type C_3 can be expressed as $C_1 \sqsubseteq C_2 \sqcap \exists rel.C_3$. More expressive TBoxes can include more general axioms having complex definitions on the left side of a subsumption. The axiom $\exists rel.C_3 \sqsubseteq C_1$ can be used to express that only components of the type C_1 can be related with the ones of the type C_3. The specific variable-free syntax was specifically chosen to make TBox statements easier to read, but most of DLs can also be translated into first-order logic formulas.

Description logics systems provide a number of sound and complete reasoning services to their customers. *Classification*, also referred to as subsumption algorithm, determines for each concept C a set of parents – the most specific concepts that subsume C – and a set of children – the most specific concepts that are subsumed by C. Computation of parents and children for each concept allows to organize them into a subsumption hierarchy, which provides important information about explicit and implicit subsumptions of concepts in TBox. *Realization*, i.e. instantiation, computes for each individual a set of most specific concepts such that this individual is an instance of. A consistency checking algorithm can determine whether a knowledge base is non-contradictory. In addition, DL systems support *query answering*, i.e. computation of all individuals that are instances of a given concept description.

Configuration has been widely addressed by DL researchers. One of the first papers [OK88] represents configuration as a consistency maintenance task of a DL knowledge base. The most prominent commercial configuration systems were developed in AT&T for telecommunication equipment [WWV+93a, MW98] and by Ford Motor Company [Ryc96].

Advantages of DL application can be clearly seen during the knowledge acquisition phase. The component types are naturally represented as concepts. Attributes of a type and relations between them are expressed as roles. The reasoning services help their customers to verify whether the formulated concept descriptions capture the intended meaning of type definition. For instance, classification can be used to organize the concepts in a hierarchy, thus, simplifying verification of the types hierarchy. Simple, variable-free syntax of DL allows to formulate axioms describing

the requirements that are easy to read and understand. The experience of exploitation of PROSE system [McG10] at AT&T showed that most of the development and maintenance of configurators can be done by people who have only little experience with DL.

Nevertheless, the main drawback of the DL systems is that standard reasoning tasks listed above are unable to capture generative aspect of configuration. That is, a DL system cannot generate a set of individuals that correspond to a solution of configuration problem. The system can only verify whether a given set of individuals is a configuration or not. Therefore, DL systems applied in configuration domain are usually embedded in an architecture providing generation facilities such as rule-based systems.

2.3.3 Specific configuration languages

A number of languages were developed specifically to represent configuration problems. In this section we review two of them, namely a graphical language based on Unified Modeling Language (UML) and the logic-based language LoCo. Both languages are very easy to learn and apply. Moreover, they can be translated into a variety of other knowledge representation languages that allow efficient computation of configurations.

Development of the UML-based language was influenced by the fact that complexity of modern configuration systems and their integration with many other software systems within an organization cause an increasing load on software developers. Model-based approaches described earlier allow to simplify the development process by a clear separation between configuration knowledge and inference algorithms used to obtain solutions. However, in many cases a knowledge representation language used to describe configuration and customer requirements is different from the modeling and programming languages used to implement other parts of a software system.

UML is one of the most widely used software modeling languages. Developed to simplify software design within object oriented paradigm, UML allows specification of static and dynamic models. The latter are used to model the behavior of a system whereas the former specify system's architecture including classes and relations between them such as hierarchies, aggregations or dependencies. The authors of [FFJ00] recognized that UML allows to abstract from a knowledge representation language of a particular configurator. Classes in UML can be used to model types of the components and the relations between types can be expressed by the corresponding relations in UML. The configuration and customer requirements that cannot be expressed as classes and their

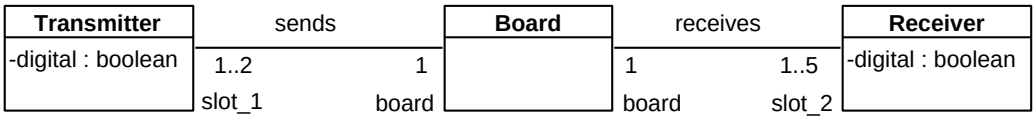

Figure 2.1: Modeling configuration problems in UML

relations can be encoded using Object Constraint Language (OCL) which is a part of the UML standard. Stereotypes, such as «ComponentType», «Resource» or «Port», can be used to characterize domain specific modeling concepts.

Reconsider the example introduced in Section 2.3.1. The component types Transmitter, Receiver and Board are represented as classes in Figure 2.1. The associations indicate that a board can receive data with at least 1 and at most 5 receivers and send with 1 or 2 transmitters. Attributes of component types Transmitter and Receiver are modeled as properties of corresponding classes. The additional constraint, namely that all receivers and transmitters installed on a board must be either digital or not, can be expressed in OCL as follows:

$$context\ Board\ inv : self.slot\_1 \rightarrow forAll(t : Transmitter \mid self.slot\_2 \rightarrow forAll(r : Receiver \mid t.digital = r.digital))$$

Application of UML as a knowledge representation language for configuration systems has many advantages, like:

- ease of integration with industrial software development processes which already rely on heavy use of UML,
- simplified exchange of configuration knowledge among experts with different background,
- possibility to translate a model into various knowledge representation languages used in different configurators.

Regarding the latter, Felfernig et al. [FFJ00] proposed a transformation from UML to a configuration language based on first-order logic which can be used by a general configuration engine to compute configurations. The authors [FH13] also show how models developed in UML can be represented as a generative CSP. The latter is very suitable for UML based development because of an object-oriented approach to the formulation of problems. Namely, classes and relations of UML are translated directly to classes and associations of a generative CSP. The OCL constraints are mapped

to constraints in CSP. Finally, the dynamic part, corresponding to creation of instances of declared classes, is done by a solver.

LoCo [ADV12] has been developed specifically to represent configuration problems. It is a subset of first-order logic with equality interpreted as identity. The language allows specification of component types and all possible relations between them required to express a configuration problem. Each component type is modeled by an n-ary predicate $Component(id, \vec{x})$ where id stands for a unique identifier of a component instance and $\vec{x}$ is a vector of constants representing attributes of a component. The terms of an atom over $Component/n$ predicate belong to different sorts such as numbers, strings, etc., and all sorts are mutually disjoint in order to guarantee that all component identifiers are unique. Moreover, for each component type there is a sort ID including all identifiers of components of that type. The relations between two component types C_i and C_j are expressed by C_i2C_j binary predicate of the sort $ID_i \times ID_j$. The relations are modeled as follows:

$$\forall id_i, \vec{x} \quad C_i(id_i, \vec{x}) \rightarrow \exists_l^u id_j \ \forall \vec{y} \quad C_j(id_j, \vec{y}) \wedge C_i2C_j(id_i, id_j) \wedge \phi(id_i, id_j, \vec{x}, \vec{y})$$

where $\phi(id_i, id_j, \vec{x}, \vec{y})$ is a subformula expressing constraints on the relation between C_i and C_j. $\exists_l^u$ is a counting existential quantifier expressing that each component of the type C_i is related to at least l and at most u components of the type C_j. A formula of the same type can be used to express the inverse part of C_i2C_j relation defining a requirement on the number of components of the type C_i needed for each C_j. For instance, the relation $sends$ presented in Figure 2.1 can be modeled as

$$\forall id_b \ Board(id_b) \rightarrow \exists_1^2 id_t \ \forall digital \ Transmitter(id_t, digital) \wedge sends(id_b, id_t)$$
$$\forall id_t, digital \ Transmitter(id_t, digital) \rightarrow \exists_1^1 id_b \ Board(id_b) \wedge sends(id_b, id_t)$$

More complex relations can also be expressed in LoCo using formulas that are extensions of the basic relation axiom given above.

The language differentiates between input and generated components types. For input component types it is assumed that the number of components to be used in a configuration is known, whereas for the generated types the number of required instances has to be determined from the provided encoding of a configuration problem. Thus, for a relation C_i2C_j between component types C_i and C_j (see Figure 2.2) LoCo solves a set of inequalities in order to determine the number of components to be generated from lower l_i and l_j bounds as well as upper u_i and u_j of counting

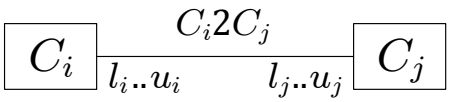

Figure 2.2: Relation between two components in LoCo

existential quantifiers as follows:

$$l_i * lb(C_i) \leq u_j * |\mathbf{C}_j| \quad \text{and} \quad l_j * |\mathbf{C}_j| \leq u_i * ub(\mathbf{C}_i)$$

where $\mathbf{C}$ is a set of components of the type C, $lb(\mathbf{C})$ and $ub(\mathbf{C})$ denote lower and upper bounds on the cardinality of the set $\mathbf{C}$. Reconsider the *sends* relation defined above and assume that the set of input components comprises one component of the type Board, i.e. $|Board| = 1$. Then for lower $lb(Transmitter)$ and upper $ub(Transmitter)$ bounds on the set of Transmitter components can be computed as:

$$1 * lb(Transmitter) \leq 2 * |Board|$$
$$1 * |Board| \leq 1 * ub(Transmitter)$$

Since one component type can have many possible relations LoCo includes an algorithm able to determine:

1. whether the number of generated components is finite, thus guaranteeing that the configuration is finite;

2. upper and lower bounds on the number of required generated components.

The first problem can be solved in polynomial time. The second one is NP-complete if tight bounds have to be determined. Therefore, LoCo includes a polynomial time algorithm computing an approximation for the upper and lower bounds.

LoCo encodings can be translated to several KRRs for which powerful solving methods exist. One of the KRR suggested in [ADV12] is answer set programming which showed good performance when applied to real-world configuration problems.

2.3.4 Answer set programming

Configuration problems can be efficiently handled by Answer Set Programming (ASP) [GL88, SN01, LPF+06, GKKS12, BET11]. The ASP paradigm has gained much attention over the past decades because it allows modeling and solving of both decision and optimization problems in a declarative way. ASP programs are defined over a function-free first-order language including at least one constant. A program Π is a finite set of normal rules of the form:

$$a \text{ :- } b_1, \ldots, b_m, \mathit{not}\ b_{m+1}, \ldots, \mathit{not}\ b_n.$$

where a and b_i are atoms, *not* denotes default negation and " :- " stands for logical implication. An *atom* is an expression of the form $p(t_1,\ldots,t_k)$, where p is a predicate and $t_1,\ldots,t_k$ are *terms*, i.e. constants and variables. An atom is *ground*, if it is variable free. A non-ground atom of a program Π can be grounded by substitution of variables with constants appearing in Π. A rule r is ground if all its atoms are ground. Finally, a program Π is ground if all its rules are ground. Hereafter, the ground instantiation of a program Π is denoted by $Gr(\Pi)$ and $At(\Pi)$ denotes the set of all ground atoms appearing in $Gr(\Pi)$. A *literal* is either an atom a (positive) or its negation *not* a (negative). The set of atoms $H(r) = \{a\}$ is called the *head* of the rule and the set of atoms $B(r) = \{b_1, \ldots, b_m, \mathit{not}\ b_{m+1}, \ldots, \mathit{not}\ b_n\}$ is the *body*. In addition, it is useful to differentiate between the sets $B^+ = \{b_1, \ldots, b_m\}$ and $B^- = \{b_{m+1}, \ldots, b_n\}$ comprising positive and negative body literals respectively. A rule with the empty body and a single atom in the head (a.) is called a *fact*, whereas a rule with empty head is called an *integrity constraint* (:- $b_1, \ldots, b_m$, *not* $b_{m+1}, \ldots,$ *not* b_n.). A set of facts constitutes an *Extensional Database* (EDB) and the remaining rules form an *Intentional Database* (IDB). A predicate that appears only in facts is called EDB predicate whereas all others are IDB predicates.

A ground atom a is true, if it is derived from some rule and *not* a is true, if it cannot be derived. A head a of a rule is derived to be true, if all literals b_i of the body are true. Derivation of the head a of facts is always justified. An answer set is a set of ground atoms $AS = \{a_1,\ldots,a_n\}$ such that the truth of each atom is justified by some rule and each rule is satisfied. The latter means that whenever $b_1,\ldots,b_m \in AS$ and $b_{m+1},\ldots,b_n \notin AS$ then $a \in AS$. Undesired answer sets can be eliminated by integrity constraints where the empty head stands for false. Consequently, the body of such rule can never be satisfied in an answer set.

Formally, the *semantics* of ASP programs is defined by Gelfond and Lifschitz [GL91] as follows: Let an *interpretation* I for Π be a set of ground atoms $I \subseteq At(\Pi)$. For a ground program $Gr(\Pi)$ and an interpretation I the Gelfond-Lifschitz reduct Π^I is defined as a set of ground rules:

$$\Pi^I = \{H(r) \leftarrow B^+(r) | r \in Gr(\Pi), I \cap B^-(r) = \emptyset\}$$

I is an *answer set* of Π, if I is a minimal model of Π^I. The program Π is *inconsistent*, if the set of all answer sets $AS(\Pi) = \emptyset$.

Extensions of ASP allow also other types of atoms. For instance, weight atom

$$l\,\{a_1 = w_1, \ldots, a_m = w_m, not\ a_{m+1} = w_{m+1}, \ldots, not\ a_n = w_n\}\,u$$

(called weight constraint) is true, if the sum S of weights w_i of all ground atoms $a_i \in AS$, $i = 1,\ldots,m$ and weights w_j of all $a_j \notin AS$, $j = m+1,\ldots,n$ is between the lower l and upper u bounds, i.e. $l \leq S \leq u$. A special case of a weight atom is a cardinality atom (also called cardinality constraint)

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

where each weight equals 1. A rule including a cardinality atom with only positive literals in its head is called choice rule, since if all literals of the body are satisfied then at least l and at most u atoms must be included in an answer set:

$$l\,\{a_1, \ldots, a_n\}\,u \text{ :- } b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n$$

For instance, the example introduced in Section 2.3.1 can be expressed in ASP as follows:

$1\{sends(B,T) : transmitter(T)\}2$:- $board(B)$.
$1\{sends(B,T) : board(B)\}1$:- $transmitter(T)$.
$1\{receives(B,R) : receiver(R)\}5$:- $board(B)$.
$1\{receives(B,R) : board(B)\}1$:- $receiver(R)$.
:- $sends(B,T), transmitterT(T,TT), receives(B,R), receiverT(R,RT), TT \neq RT$.

The first two rules are used to specify that each board can send signals with at least 1 and at most 2

transmitters and each transmitter is connected to 1 board only. Analogously, the relation between a board and receivers is modeled. The last rule is an integrity constraint requiring installed transmitters and receivers to be of the same type.

Normal rules as well as rules including weight and cardinality atoms were used in the first application of ASP to configuration problems [SNTS01]. Configuration and customer requirements in [SNTS01] are split into two parts: ontological definitions and configuration model. The former are constant for all possible requirements and include basic definitions of component types, resources and relations. The configuration model represents domain-specific knowledge in terms of predefined ontological definitions. That is, one has to provide available components and relations between them using atoms over ontological predicates. The components are mostly represented as facts and relations are choice rules, where upper and lower bounds of a cardinality atom represent the cardinality of the relation. Once both parts are given, the computation of configurations can be done by an ASP solver.

The further extension of the approach presented by Soininen et al. [SNTS01] can be found in [THAS13]. The authors suggest a high-level object oriented modeling language (Product Configuration Modeling Language) and a web-based graphical customer interface used to simplify the modeling of configuration and customer requirements. In addition, the paper shows how approximate reasoning methods for ASP based on well-founded semantics [LRS97, SNS02] can be used to speed up the computations such that the approach can be used in on-line settings for real-world problems. Recently a framework for describing object-oriented knowledge bases using answer set programming was presented in [SFRF13]. The authors suggested a general mapping from an object-oriented formalism (UML) to ASP. The framework can be applied to model different (re)configuration scenarios occurring in practice of Siemens.

2.4 Reconfiguration

So far we provided the description of the configuration problem and the most popular approaches to solve it. All of them assume that we are modeling the configuration problem from scratch giving a set of configuration components and constraints which define valid solutions. However, when we do not create a configuration from scratch, i.e. some parts of an existing configuration have to be adapted, we are faced with a problem which is called *reconfiguration*. This is an important activity in the after-sales life-cycle for companies selling configurable products or services. Typically such

products have a long life time and configuration requirements for them are changing in parallel with the customers' business [KR99, Man05, FH13]. For example, if some components used in a product are not produced any more, new functionality has to be added to a system, etc. In this case, some relations between new and existing components have to be created or some of the existing relations have to be changed in order to meet modified configuration requirements. Some typical challenges occurring whenever a configuration is changed are mentioned by Falkner and Haselböck in [FH13]:

- add new component types, attributes, associations;
- delete outdated component types, attributes, associations;
- change, add, delete constraints;
- change attributes types, cardinalities of associations, etc.

Standard techniques used in configuration systems might not be suitable when a knowledge base evolves over time. New methods are required to handle the transformation of a legacy configuration. For example, it can include such actions as deletion of some components or connections between them or creation of new components and relations. These actions are usually associated with domain-specific costs which reflect preferences among possible reconfiguration solutions. Of course, different reconfiguration solutions can be found depending on these modification costs. Therefore, the optimization criteria of a configuration problem have to be extended with additional criteria defined for the reconfiguration costs. Moreover, depending on the domain-specific requirements reconfiguration costs may possess a higher priority than of the configuration costs.

Modern approaches to reconfiguration can be classified into *revision-based* [MSTS99, SMSS03], *model-based* [CR91, SW98b] or *combined* [FRF+11a]. The *revision-based* approaches identify a solution of the reconfiguration problem as a sequence of transformations of the legacy configuration required to obtain the target configuration. The transformations can be represented as delete and add actions applied to components and connections in the legacy configuration. The application of each action to a configuration during the transformation process is determined by action's preconditions. Both actions and preconditions are stored in a knowledge base describing possible causes of faults and their repairs. In practice, revision-based approaches are successfully applied to evolution of knowledge bases such as ontologies [FMK+08, OFSA11]. In particular, Stojanovic et al. [SMSS03] showed that evolution of ontologies can be represented as a reconfiguration problem.

The first attempt to use *model-based approach* was published by Crow and Rushby [CR91] back in 1991. In their approach authors suggest to view the reconfiguration as an extension of Reiter's theory of diagnosis [Rei87]. That is, the legacy configuration has to be modified if it is inconsistent with modified customer or configuration requirements. In order to restore the consistency some of the assumptions of a legacy configuration are faulty and must be removed or modified. In the diagnosis theory possible sets of faulty assumptions correspond to a set of diagnoses and, therefore, algorithms suggested in [Rei87] can be applied to the reconfiguration problems. More recent diagnostic engines suggested by Stumptner and Wotawa in [SW98b] allow specification of additional information, such as invariant components and connections of a legacy configuration, costs of reconfiguration actions or fault probabilities for legacy components, in order to determine a solution preferred by a customer.

Combined approach suggested by Friedrich et al. in [FRF+11a] uses non-monotonic features of ASP to simulate the diagnosis process and is able to determine the set of actions required to obtain a solution of a reconfiguration problem similarly to revision-based approaches (see Chapter 3). Particularly, the approach allows to encode invariants of a legacy configuration, i.e. components or relations that have to be reused in a reconfiguration solution, as well as to define the parts of the system that can be changed, i.e. either deleted or reused in the system. The set of modified customer and configuration requirements allow to determine a set of repair actions that must be executed in order to extend a partial configuration, resulting in deletion of components and/or connections, to a reconfiguration solution. Moreover, the preferred solution can be computed using reconfiguration costs given by a customer, e.g. minimizing the overall costs of all suggested modifications. This approach is presented in the following chapter in more details.

CHAPTER

3 Knowledge-based (re)configuration using ASP

In this chapter we present a logical formalism allowing a unified representation and solution of both configuration and reconfiguration problems. As it was discussed in Chapter 2 different knowledge representation and reasoning approaches exist which are able to capture configuration model and construct (re)configurations. Because of the remarkable advances of ASP [GL88, SNS02, LPF+06, BET11, GKKS12] we selected this logical formalism as a basis of our knowledge representation language designed for a compact description of (re)configuration problems. In addition, generation of optimized (re)configurations can be done using standard ASP solvers.

Following the knowledge-based approach the customer and configuration requirements of a configuration problem are encoded as a set of ASP rules. The configuration program comprises two parts: (i) extensional database (EDB) including facts describing the available component catalog which corresponds to the customer requirements and (ii) intentional database (IDB) which is a set of rules encoding the configuration requirements. A configuration is defined as a subset of a minimal logical model of an ASP program.

Reconfiguration is an important activity for companies selling configurable products or services which have a long life time. However, identification of a set of necessary modifications of a legacy configuration is a hard problem, since even small changes in the requirements might imply significant discrepancies between legacy and requested configurations. In our approach the first stage of a reconfiguration case is similar to the configuration one. That is, a knowledge engineer designs a program encoding a new configuration problem according to the new customer and configuration requirements. In the next step, the extensional part of a knowledge base is altered with the facts describing a legacy configuration and a set of transformation rules is added to the intentional part.

The latter are used to describe allowed modifications of the legacy configuration which can be done in order to obtain a new modified configuration. In terms of ASP the transformation rules map facts describing the legacy configuration to facts of the new configuration problem. In order to solve a reconfiguration problem an ASP solver has to decide which parts of the legacy configuration can be reused or deleted in the new configuration and which new parts have to be created.

In this chapter in Section 3.1 we present an introductory example of a configuration problem and sample reconfiguration scenarios. Then, configuration problems are defined in Section 3.2. In Section 3.3 an overview of the basic ASP concepts is given which is followed by the exemplification of the modeling in Section 3.4. Section 3.5 provides the definition of reconfiguration problems. Subsequently, modeling patters and an example of their application are provided in Section 3.6.

3.1 (Re)configuration example

The House problem is a simple yet important problem exemplifying different configuration and reconfiguration scenarios. This problem is an abstraction of several configuration problems occurring in practice, i.e. entities may be contained in other entities but some restrictions must be fulfilled. Consider a configuration case in which a set of things owned by a number of persons must be stored in cabinets which are then placed in rooms of a house. The problem description comprises such concepts as *person*, *thing*, *cabinet*, and *room* where persons are related to things, things are related to cabinets, cabinets are related to rooms, and rooms are related to persons. These relations are modeled either by roles, associations, or predicate symbols depending on the modeling language (e.g. description logic, UML, or predicate logic).

Problem 1 (House configuration problem)**.** Let *person* be a set of persons, *thing* be a set of things and *person2thing* be a set of tuples $\langle p_i, t_j \rangle$ denoting the ownership relation between persons and things. Moreover, let *cabinet* and *room* be the sets of available cabinets and available rooms, respectively.

The problem is to find the sets $cabinet' \subseteq cabinet$ and $room' \subseteq room$ such that assignments of things to cabinets, denoted *cabinet2thing*, and cabinets to rooms, denoted *room2cabinet*, fulfill the following requirements:

- each thing belongs to only one person;
- each thing must be stored in exactly one cabinet;

- a cabinet can contain at most 5 things;
- every cabinet can be placed in exactly one room;
- a room can contain at most 4 cabinets;
- every room can belong only to one person;
- and a room may only contain cabinets storing things of the owner of the room.

The assignments correspond to sets of tuples, namely, $cabinet2thing = \{\langle c_1, t_1\rangle, \ldots, \langle c_n, t_m\rangle\}$ and $room2cabinet = \{\langle r_1, c_1\rangle, \ldots, \langle r_l, c_n\rangle\}$, where $t_j \in thing$, $c_i \in cabinet'$ and $r_k \in room'$.

The solution is *optimal* iff the sets $cabinet'$ and $room'$ comprise minimal number of elements, i.e. there is neither a set $cabinet'' \subset cabinet'$ nor a set $room'' \subset room'$ for which assignments *cabinet2thing* and *room2cabinet* exist that satisfy all the requirements defined above.

The tuple $\langle person, thing, person2thing, cabinet, room\rangle$ defines an *instance* of the House configuration problem.

An input to the configuration problem, i.e. sets of persons, things and the ownership relation between persons and things, is provided by a user. This input corresponds to the *customer requirements* since they reflects the individual needs of a customer using a configuration system. The *configuration requirements* specify the properties of a system to be configured. In the case of the House problem these are the requirements to the assignments of things to cabinets and cabinets to rooms. In order to keep the example simple we only consider configuration of one house and represent all individuals using unique integer identifiers. Informally, a configuration is every instantiation of the relations which satisfies all constraints.

Let a sample house problem instance include two persons such that the first person owns five things numbered from 3 to 7 and the second person owns one thing 8. A solution to this house

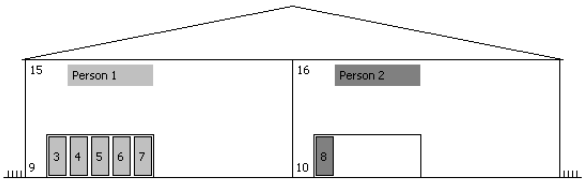

Figure 3.1: Solution of the sample house configuration problem

configuration problem instance is shown in Figure 3.1 where the house configuration includes rooms 15 and 16, two cabinets 9 and 10, and six things numbered from 3 to 8.

Reconfiguration is necessary, whenever the customer requirements or configuration requirements are changed. For instance, it becomes necessary to differentiate between long and short things.

Problem 2 (House reconfiguration problem)**.** Given set of persons *person*, a set of things *thing*, a set of tuples denoting ownership relations between persons and things *person2thing*, the sets of cabinets *cabinet* and rooms *room*. Let $\langle person_L, thing_L, person2thing_L, cabinet_L, room_L, cabinet2thing_L, room2cabinet_L \rangle$ be a legacy solution of an instance of the House configuration problem (Problem 1). In addition, let $thingLong \subseteq (thing \cup thing_L)$ denote a set of things that are considered as long. All other things are considered as short. Moreover, all legacy cabinets $c \in cabinet_L$ are initially considered as small.

The problem is to find a set of cabinets $cabinet' \subseteq (cabinet \cup cabinet_L)$, a set of high cabinets $cabinetHigh \subseteq cabinet'$ and the set of rooms $room' \subseteq (room \cup room_L)$ such the sets of assignments *cabinet2thing* and *room2cabinet* satisfy both the requirements of Problem 1 and the following new requirements:

- a cabinet $c \in cabinet'$ is either small or high;
- a long thing $t \in thingLong$ can only be put into a high cabinet $c \in cabinetHigh$;
- a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room.

The assignments are sets of tuples $cabinet2thing = \{\langle c_1, t_1 \rangle, \ldots, \langle c_n, t_m \rangle\}$ and $room2cabinet = \{\langle r_1, c_1 \rangle, \ldots, \langle r_l, c_n \rangle\}$, where $t_j \in (thing \cup thing_L)$, $c_i \in cabinet'$ and $r_k \in room'$.

Let $f(\cdot)$ be a costs function that maps a solution of a reconfiguration problem to a positive integer. The solution is *optimal* iff it minimizes the costs function $f(\cdot)$.

In the case of reconfiguration problem, the customer requirements are extended with a definition for each thing whether it is long or short. For instance, for the previously given house example the customer provides information that the things 3 and 8 are long; all others are short. Moreover, the first person gets an additional long thing 21. The changes to the legacy configuration are summarized in Figure 3.2 showing an inconsistent configuration, where thing 21 is not placed in any of the cabinets, and cabinets 9 and 10 are too small for things 3 and 8.

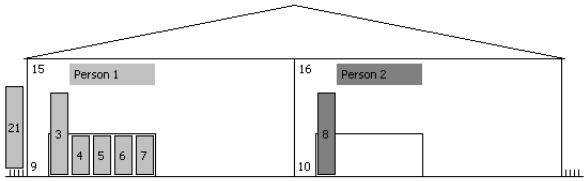

Figure 3.2: House reconfiguration initial state

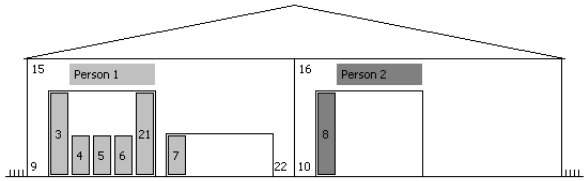

Figure 3.3: House reconfiguration solution 1

To obtain a solution which is shown in Figure 3.3 the reconfiguration process changes the size of cabinets 9 and 10 to high and puts the new thing 21 into cabinet 9. A new small cabinet 22 is created for thing 7.

In the reconfiguration process every modification to the existing configuration (reusing, deleting and creating individuals and their relations) is associated with some cost. Therefore, the reconfiguration problem is to find a consistent configuration by removing the inconsistencies and minimizing the costs involved. Different solutions will be found depending on the given modification costs specified by a customer. If, for example, the costs for adding a new high cabinet are less than the cost for changing an existing small cabinet into a high cabinet, then the previous solution should be rejected as its costs are too high. One of the solutions with less reconfiguration costs (see Figure 3.4) includes two new cabinets 22 and 23, because the creation of new cabinets is cheaper than converting the existing small cabinets into high cabinets. Also it contains the empty cabinet 10, because it's cheaper to keep the cabinet than to delete it. Note, this behavior can be controlled by

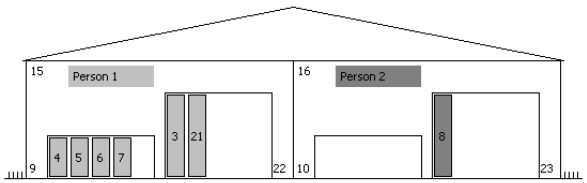

Figure 3.4: House reconfiguration solution 2

the domain specific costs.

3.2 Configuration problems

We employ a definition of configuration problems based on logical descriptions [SNTS01, FFJS04]. The idea is that every finite Herbrand-model contains the description of exactly one configuration.

The description of a configuration is defined by relations expressed by a set of predicates P_S. This set of predicates is called *solution schema*. For our example the solution schema consists of four unary predicates *thing*/1, *person*/1, *cabinet*/1 and *room*/1 representing the individuals and four binary predicates, namely *personTOthing*/2, *personTOroom*/2, *roomTOcabinet*/2 and *cabinetTOthing*/2 representing the relations. An instantiation of this solution schema corresponds to a configuration and its fragment is given below. Note, this description of a configuration generalizes the component/port models or variable/value based descriptions of a configuration.

person(*1*). *thing*(*3*). *room*(*15*). *cabinet*(*9*).
cabinetTOthing(*9,3*). *personTOthing*(*1,3*).
roomTOcabinet(*15,9*). *personTOroom*(*1,15*).
. . .

We assume that every predicate symbol is unique in a logical theory and has a unique arity. The set of Herbrand-models is specified by a set of logical sentences REQ which usually comprises individual *customer requirements* and *configuration requirements*. Configuration requirements reflect the set of all allowed configurations for an artifact, whereas customer requirements may comprise facts and logical sentences specifying the individual needs of customers. The same configuration requirements are a basis for different sets of customer requirements. For instance, the component library of a technical system is stable for some time.

Definition 3 (Configuration problem instance)**.** A configuration problem instance $\langle REQ, P_S \rangle$ is defined by a set of logical sentences REQ representing requirements and P_S a set of predicate symbols representing the solution schema. For optimization purposes an objective function $f(CON) \mapsto \mathbb{N}$ maps any set of atoms CON to positive integers where CON contains only atoms whose predicate symbols are in P_S.

Let $HM(LS)$ denote the set of Herbrand-models of a set of logical sentences LS for a given semantics.

Definition 4 (Configuration)**.** CON is a configuration for a configuration problem instance $CPI = \langle REQ, P_S \rangle$ iff there is a Herbrand-model $M \in HM(REQ)$ and CON is the set of all the elements of M whose predicate symbols are in P_S and CON is finite, i.e. $CON = \{p(\bar{t}) | p \in P_S \text{ and } p(\bar{t}) \in M)\}$. By $p(\bar{t})$ we denote a ground instance of p with a term vector $\bar{t}$.
CON is an optimal configuration for *CPI* iff CON is a configuration for *CPI* and there is no configuration CON' of *CPI* such that $f(CON') < f(CON)$.

Definition 5 (Configuration problems)**.** Let the instances of configuration problems be defined by $\langle REQ, P_S \rangle$ and objective functions $f(\cdot)$.
Decision problem: Given a set of atoms CON, decide if CON is a configuration for a configuration problem instance.
Generation (optimization) problem: Generate a set of atoms CON such that CON is a(n optimal) configuration for a configuration problem instance $CPI = \langle REQ, P_S \rangle$.

The set of Herbrand-models depends on the semantics of the employed logic. We apply answer set programming and a stable model semantics for knowledge representation and reasoning, because this approach allows a concise and modular specification, ensures decidability, and avoids the inclusion of unjustified atoms (e.g. unjustified components) in configurations [SNTS01].

3.3 Answer set programming overview

Answer Set Programming is an approach to declarative modeling and solving hard computational problems using model-based problem specification methodology. This paradigm is a result of intensive research in the areas of knowledge representation, deductive databases and logic programming. In general, ASP is a subset of first-order logic interpreted under stable model semantics [GL88] and extended with default negation, aggregation, and optimization.

ASP is a relatively new paradigm which, nevertheless, has a number of successful applications. Such examples can be found in a team building system used to configure and plan shifts at Gioia-Tauro Seaport [RGA+12] or repair handling of configurable Web-service workflows [FFM+10]. Another applications of ASP such as decision support system for the space shuttle [NBG+01] and detection of inconsistencies in large biological networks [GGI+10] are only few examples of its deployment. The prototypes of pure product configuration tools ready for industrial application were developed by e.g [SNTS01, THAS13, SFRF13].

In this section we give an overview of ASP and language constructs as needed, for details see e.g. [GL88, LPF+06, EIK09, BET11, GKKS12]. An ASP program is a finite set of rules of the form:

$$a \text{ :- } b_1,\ldots,b_m,, \textit{not } b_{m+1},\ldots,\textit{not } b_n. \tag{3.1}$$

where a and b_i are atoms of the form $predicate(term_1,\ldots,term_n)$, *not* denotes default negation and " :- " denotes the logical implication. A *term* is either a variable or a constant[1]. An atom is ground, if it is variable free. A non-ground atom can be grounded by substituting all variables with constants appearing in the program. In most of ASP languages variables are denoted by strings starting with uppercase letters and constants as well as predicates by strings starting with lower case letters. An atom together with its negation is called *literal*, e.g. a is a positive and *not a* is a negative literal. In the rule (3.1) the literal a is the *head* of the rule and the conjunction $b_1,\ldots,b_m,\textit{not } b_{m+1},\ldots,\textit{not } b_n$ is the *body*. A rule with an empty head, standing for *false*, is called an *integrity constraint*, i.e. every interpretation that satisfies the body of the constraint is not an answer set (is not a configuration solution). A rule with an empty body is called a *fact*.

Rule (3.1) derives that the atom a in the head of the rule is *true* if all literals of the body are *true*, i.e. there is a derivation for each positive literal $b_1,\ldots,b_m$ whereas none of the atoms of the negative literals *not* b_{m+1}, ..., *not* b_n can be derived.

In the following sample program in which atoms contain no variables and, therefore, correspond to propositional symbols:

cabinet(*c1*).
cabinetSmall(*c1*) :- *cabinet*(*c1*), *not cabinetHigh*(*c1*).

we can derive that an individual $c1$ is of the type *cabinetSmall*, if we establish that both $c1$ is of the type *cabinet* and there is no evidence that $c1$ is of the type *cabinetHigh*. That is, we can derive *cabinetSmall*(*c1*), if we extend the program with the fact *cabinet(c1)*. Hence the program justifies only one set {*cabinetSmall*(*c1*). *cabinet*(*c1*).}.

Processing of a general ASP program P, in which atoms can contain variables, is done in two stages as shown in Figure 3.5. First the program is grounded, i.e. P is replaced by a possibly small equivalent propositional program $grnd(P)$ in which all atoms are variable-free. Modern grounders, such as GRINGO [GKO+09] or DLV [LPF+06], apply intelligent grounding methods including partial

[1]In some ASP languages, such as GRINGO [GKO+09], a term can also be an uninterpreted function symbol.

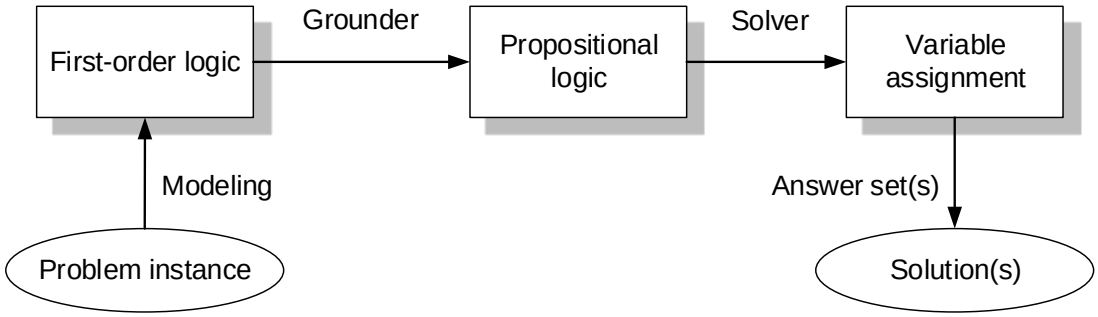

Figure 3.5: Modeling stages using ASP

evaluation and program rewriting to reduce the size of ground programs. In the second stage an ASP solver is applied to the ground program, which identifies all answer sets (see [BET11] for an introduction to computation of answer sets). The several ASP solvers are available which compete in the biennial ASP competitions, for example, CLASP from POTASSCO ASP suite[2], DLV[3] and SMODELS systems[4].

Following the definition of configuration problems based on logical descriptions, presented in [SNTS01, FFJS04], each configuration is a subset of a finite Herbrand-model. Given the stable model semantics used in ASP, a Herbrand interpretation I is a *model* of a program P *iff* (a) it satisfies all the rules in P, (b) for every atom $a_i \in I$ there exists a justification based on given facts and (c) I is minimal under set inclusion among all (consistent) interpretations. Hence the set of finite Herbrand-models corresponds to a set of answer sets of a program P. This allows us to use state-of-the-art ASP solvers to solve a variety of configuration problems.

In this thesis we use an ASP dialect implemented in GRINGO 3 [GKO+09] which is a part of POTASSCO ASP collection [GKK+11b] and includes a number of extensions simplifying the presentation of the programs. Thus, it allows definition of *weight constraints* which are defined as $l[a_1 = w_1, \ldots, a_n = w_n]u$ where a_i are atoms, w_i are weights of the atoms and l, u are integers specifying lower and upper bounds. Such constraints allow declaration of choices, i.e. such number of atoms from the set $\{a_1, \ldots, a_n\}$ must be true that the sum of corresponding weights is between l and u. If the lower or upper bounds are missing, then the ASP grounder substitutes $l = 0$ and $u = n$,

[2] http://potassco.sourceforge.net
[3] https://www.mat.unical.it/dlv-complex
[4] http://www.tcs.hut.fi/Software/smodels

where n is the sum of the weights of all atoms in the set. A special case of the weight constraints defined by curly brackets are *cardinality constraints* where each weight $w_i = 1$ and duplicated literals are removed.

ASP frameworks such as Potassco [GKK+11b] include operators that are used in order to generate sets of atoms. The *range* operator (". .") is used to generate a set of atoms such that each atom includes one of the integer constants from a given range of integers. The *generate* operator (":") is used in weight constraints to create sets of atoms used in it.

In order to allow logical variables and functional symbols but to guarantee decidability the set of allowed rules is restricted. The version of Potassco used in this thesis requires programs to be *safe* [GKK+11b]. A rule is safe if each variable appearing in an atom of this rule also appears in at least one positive atom b_i of the body of that rule, where $i = 1, \ldots, m$ and b_i is not a comparative build-in [LPF+06]. A program is safe if its each rule is safe.

Example Assume that we want to encode a simple problem instance of the House example introduced in Section 3.1 including six things and two cabinets. These customer requirements can be represented as facts:

cabinet(*9. .10*). *thing*(*3. .8*).
personTOthing(*1,3*).
. . .

Note, the atom *cabinet*(*9. .10*). is equivalent to the set of atoms {*cabinet*(*9*). *cabinet*(*10*).}

Cardinality constraints can be efficiently used to encode relations between configuration component types. The relation between things and cabinets, i.e. that each thing must be placed in exactly one cabinet, can be encoded using the following choice rule:

1{*cabinetTOthing*(*X,Y*) : *cabinet*(*X*)}*1* :- *thing*(*Y*).

When the rule above is grounded, the grounder generates six rules - one for each thing:

1{*cabinetTOthing*(*10,3*),*cabinetTOthing*(*9,3*)}*1* :- *thing*(*3*).
. . .
1{*cabinetTOthing*(*10,8*),*cabinetTOthing*(*9,8*)}*1* :- *thing*(*8*).

In order to allow at most five things to be put in a cabinet, we add the following cardinality constraint to our program:

$:\text{-}\ cabinet(X),\ 6\ \{cabinetTOthing(X,Y)\ :\ thing(Y)\}.$

Intuitively, due to the cardinality constraint every configuration (answer set) containing a cabinet with more than 5 things will be eliminated.

The identification of the preferred configuration solution can be done using the built-in optimization functionality of ASP solvers [SNS02, LPF+06, GKK+11b]. In the ASP dialect used in this work the optimization is defined on a weighted set of true atoms and indicated via *#minimize* or *#maximize* statements. In particular, the statement below allows minimization.

$$\#minimize[a_1 = w_1@p_1, \ldots, a_n = w_n@p_n].$$

The minimization statement is similar to the weight constraints with a possibility to assign a priority level p_i to each weighted literal. An answer set is optimal iff the sum of the weights of literals which are satisfied in this answer set is minimal (maximal) among all answer sets of a given program. Optimization is performed with respect to customer preferences expressed in the program and in the order of priorities starting from the highest priority value.

Modeling of configuration knowledge in ASP requires definition of component types and attributes, associations between them, generalization types and additional configuration constraints. As it was shortly shown above, all the definitions can be encoded in an ASP program in a very precise and clear declarative manner. In the following section modeling patterns will be discussed in more details.

3.4 Defining configuration problem instances

In [SNTS01] various modeling patterns based on weight constraints were introduced. A fixed set of ground facts define the individuals which are employed for a configuration. This fixed set of ground facts in conjunction with the level-restriction place an upper bound on the size of the number of ground rules and therefore decidability is guaranteed. At the current state of research such an upper bound on the number of individuals is necessary for many applications. In particular, it is known from database theory that so called tuple generating dependencies lead to undecidability even under rather strict syntactical restrictions [CGK08]. A tuple generating dependency is $\forall\overline{X}\forall\overline{Y}\phi(\overline{X},\overline{Y}) \rightarrow \exists\overline{Z}\psi(\overline{X},\overline{Z})$ where $\phi(\overline{X},\overline{Y})$ and $\psi(\overline{X},\overline{Z})$ are conjunctions of atoms and $\overline{X},\overline{Y}$, and $\overline{Z}$ are representing vectors of logical variables, see Section 5.1.1 for details. Unfortunately, such rules may occur in

configuration problem instances. For instance, if a condition holds, a specific individual of some type must exist and this individual must be connected to some other individuals.

However, in many cases it is undesirable to consider only a fixed number of individuals employed in a configuration. Guessing the right number of individuals for configuration generation problems or optimization problems is quite hard and often impossible. Therefore, we apply the following modeling pattern.

Let *pLower* and *pUpper* represent the upper and lower number of individuals of type p. Such a type is called *bounded*. We require each individual of a configuration, represented by its unique identifier, to be a member of exactly one bounded type. To each bounded type a domain *pDomain* is associated, representing the set of possible individuals of the bounded type. We employ numbers as identifiers, starting from some offset. For every bounded type p we add the following axioms:

$pDomain(pOffset + 1\,..\,pOffset + pUpper).$
$pLower\{p(X) : pDomain(X)\}pUpper.$

The first rule instantiates the maximal required number of unique individuals of p in *pDomain*. The second rule makes sure that at least *pLower*, but at most *pUpper* individuals of p are asserted. By these rules the required number of p individuals are asserted, in order to find a configuration within the given upper and lower bounds.

For some bounded types, e.g. $person/1$ and $thing/1$ the bounds *pLower* and *pUpper* coincide because the exact number of individuals employed in any configuration is known. In this case the fixed set of p facts can be asserted without using the rules presented above.

In our example the customer provides a number of requirements for a configuration that include definitions of *person* and *thing* individuals as well as their relations.

$person(1\,..\,2).\ thing(3\,..\,8).$
$personTOthing(1,3).\ personTOthing(1,4).$
$personTOthing(1,5).\ personTOthing(1,6).$
$personTOthing(1,7).\ personTOthing(2,8).$

For the bounded type *cabinet* we add the following rules. The upper and lower numbers of cabinets are computed based on the number of things and persons.

$cabinetDomain(9\,..\,14).$
$2\{cabinet(X) : cabinetDomain(X)\}6.$

The rules for *room* individuals are defined accordingly.

Cardinality restrictions given in Section 3.1 are encoded with cardinality constraints, where one direction of an association is encoded as a generation (choice) rule (see Section 3.3) and the other direction as a constraint. Such encoding corresponds to Guess/Check/Optimize methodology described in [LPF+06]. Note, the cardinality constraints just as the weight constraints require that logical variables appear in domain predicates. Therefore, we have to use *pDomain* predicates rather than *p* predicates, e.g. *cabinetDomain*(*X*) instead of *cabinet*(*X*). However, individuals employed in relations must also be contained in the corresponding types (see the last four rules of the next sequence of rules). By these rules we avoid situations where an individual is used in a relation, but is not included in the bounded type. In our example, if the program asserts *cabinetTOthing*(*14,1*), then *cabinet*(*14*) is also asserted.

```
1{cabinetTOthing(X,Y) : cabinetDomain(X)}1 :- thing(Y).
:- 6{cabinetTOthing(X,Y) : thing(Y)}, cabinet(X).
1{roomTOcabinet(X,Y) : roomDomain(X)}1 :- cabinet(Y).
:- 5{roomTOcabinet(X,Y) : cabinetDomain(Y)}, room(X).

room(X) :- roomTOcabinet(X,Y).
room(Y) :- personTOroom(X,Y).
cabinet(X) :- cabinetTOthing(X,Y).
cabinet(Y) :- roomTOcabinet(X,Y).
```

The next rules describe the fact that a room may contain things of its owner only.

```
personTOroom(P,R) :- personTOthing(P,X), cabinetTOthing(C,X), roomTOcabinet(R,C).
:- personTOroom(P1,R),personTOroom(P2,R), P1 != P2.
```

In addition, optimization can be applied to generate optimal configurations which minimize the overall configuration costs depending on the objective function. We model the objective function by assigning to each atom in a configuration CON some costs. This can be achieved with the following modeling pattern. By the atom $cost(create(a,w))$ the costs of creating an element a in a configuration are defined, where a is an element of CON and w is an integer. We employ the conjunction of atoms $\alpha(\overline{X},\overline{Y},W)$ to allow case specific determination of costs. For each $p \in P_S$ we include axioms of the following form in REQ: $cost(create(p(\overline{X})),W)$:- $p(\overline{X}),\alpha(\overline{X},\overline{Y},W)$ such that for each atom $p(\bar{t})$ in CON the answer set contains an atom $cost(create(p(\bar{t}),w))$ where w is

an integer. For example:

roomCost(*5*). *personTOroomCost*(*1*).
cost(*create*(*room*(*X*)),*W*) :- *room*(*X*), *roomCost*(*W*).
cost(*create*(*personTOroom*(*X,Y*)),*W*) :- *personTOroom*(*X,Y*), *personTOroomCost*(*W*).

All other costs are expressed in the same way. The sum of all costs can be minimized by means of the following optimization statement:

#minimize[*cost*(*X,W*)=*W*].

For the given example the solver finds an optimal configuration including two cabinets and two rooms with the overall cost 40 (depicted in Figure 3.1).

cabinet(*10*). *cabinet*(*9*). *room*(*16*). *room*(*15*).
roomTOcabinet(*15,9*). *roomTOcabinet*(*16,10*).
cabinetTOthing(*10,8*). *cabinetTOthing*(*9,7*). *cabinetTOthing*(*9,3*).
. . .

3.5 Reconfiguration problems

We view reconfiguration as a new configuration-generation problem where parts of a *legacy configuration* are possibly reused. The conditions under which some parts of the legacy configuration can be reused and what the consequences of a reuse are, is expressed by a set of logical sentences T which relate the legacy configuration CON and the new configuration problem instance $\langle REQ_R, P_R \rangle$.

Definition 6 (Reconfiguration problem instance)**.** A tuple $\langle \langle REQ_R, P_R \rangle, CON, T \rangle$ is a reconfiguration problem instance which is defined by $\langle REQ_R, P_R \rangle$ an instance of a configuration problem, CON a legacy configuration and T a set of logical sentences representing the transformation constraints regarding the legacy configuration.
For optimization purposes an objective function $g(CON, R) \mapsto \mathbb{N}$ maps legacy configurations CON and configurations R of $\langle REQ_R, P_R \rangle$ to positive integers.

Note, the two-placed objective function expresses the fact that the costs of a reconfiguration depend not only on the elements contained in a reconfiguration, but also on the reuse or deletion of elements of the legacy configuration.

In order to avoid name conflicts between the entities of the legacy configuration CON and instances of new configuration problems $\langle REQ_R, P_R \rangle$, we formulate P_R and REQ_R using constants not employed in CON. In particular, we use different name spaces for terms referencing individuals. Together with the unique name assumption this implies that individuals of the legacy configuration and new individuals introduced by the reconfiguration problem are disjoint.

Reconfigurations are defined analogously to configurations as a finite subset of Herbrand-models.

Definition 7 (Reconfiguration)**.** R is a reconfiguration for a reconfiguration problem instance $RCI = \langle \langle REQ_R, P_R \rangle, CON, T \rangle$ iff there is a Herbrand-model $M \in HM(REQ_R \cup CON \cup T)$ and R is the set of all the elements of M whose predicate symbols are in P_R and R is finite.
R is an optimal reconfiguration for RCI iff R is a reconfiguration for RCI and there is no reconfiguration R' of RCI such that $g(CON, R') < g(CON, R)$.

Reconfiguration problems are formulated analogously to configuration problems.

Definition 8 (Reconfiguration problems)**.** The instances of reconfiguration problems are defined by a tuple $\langle \langle REQ_R, P_R \rangle, CON, T \rangle$ and objective functions $g(\cdot, \cdot)$.
Decision problem: Given a set of atoms R, decide if R is a reconfiguration for a reconfiguration problem instance.
Generation (optimization) problem: Generate a set of atoms R such that R is a(n optimal) reconfiguration for a reconfiguration problem instance $RCI = \langle \langle REQ_R, P_R \rangle, CON, T \rangle$.

3.6 Defining reconfiguration problem instances

In the following we show typical formalization patterns and apply them to our example. The set of atoms $\{legacyConfig(a) | a \in CON\}$ describes the atoms of the legacy configuration CON. To facilitate a concise description of the problem we introduce the predicate $legacyConfig/1$ to allow quantification over the elements of the legacy configuration.

For the transformation sentences T we employ the following general patterns. For reusing parts of the legacy configuration the problem solver has to make the decision either to *reuse* or to *delete*. This is expressed by $reuse(a)$ and $delete(a)$ atoms where a is an element of CON. For each atom $a \in CON$ either $reuse(a)$ or $delete(a)$ must hold. Based on these atoms additional configuration constraints can be defined which describe the proper reuse or deletion of a part of the legacy configuration represented by atom a. In our case, reusing an atom a of the legacy configuration implies the

assertion of this atom, whereas deletion requires that the atom is not asserted. In addition, costs are associated to each $reuse(a)$ or $delete(a)$ operation. This is expressed by the atom $cost(reuse(a), w)$ or $cost(delete(a), w)$ where a is an element of CON and w is an integer specifying the corresponding costs. Furthermore, we require that in each model which contains $reuse(a)$ or $delete(a)$ also $cost(reuse(a), w)$ or $cost(delete(a), w)$ is contained in order to have defined reuse or deletion costs. The conjunctions $\beta(\overline{X}, \overline{Y}, W)$ and $\gamma(\overline{X}, \overline{Y}, W)$ are employed to define domain specific costs.

For each $p \in P_S$ the following axioms in T are included:

$1\{reuse(p(\overline{X})),\ delete(p(\overline{X}))\}1 \ :\text{-}\ legacyConfig(p(\overline{X})).$
$p(\overline{X}) \ :\text{-}\ reuse(p(\overline{X})).$
$:\text{-}\ p(\overline{X}),\ delete(p(\overline{X})).$
$cost(reuse(p(\overline{X})),\ W) \ :\text{-}\ reuse(p(\overline{X})),\ \beta(\overline{X},\overline{Y},W).$
$cost(delete(p(\overline{X})),W) \ :\text{-}\ delete(p(\overline{X})),\ \gamma(\overline{X},\overline{Y},W).$

Analogously to configuration problems, we require each individual contained in a reconfiguration to be a member of exactly one bounded type. Consequently, individuals of the legacy configuration have to be a member of the domain $pDomain(X)$ of a bounded type p of $\langle REQ_R, P_R \rangle$, because these individuals can be part of a reconfiguration through reuse. That is, there are rules of the form

$pDomain(X) \ :\text{-}\ legacyConfig(q(\ldots,X,\ldots)).$

where q is predicate symbol of the solution schema of the legacy configuration.

As for configuration problems, the number of individuals of a bounded type p is limited. For every bounded type p we add the following axioms:

$pLower\{p(X) \ : pDomain(X)\}pUpper.$

The newly generated individuals are specified by $pDomainNew/1$ for the bounded type p and we use $pNewOffset$ to generate new identifiers:

$pDomainNew(pNewOffset + 1\ldots pNewOffset + pUpper).$
$pDomain(X) \ :\text{-}\ pDomainNew(X).$

In our example, the reconfiguration problem consists of additional customer and configuration requirements described in Section 3.1. The solution schema for the reconfiguration problem is an extension of the solution schema of the original configuration problem by $cabinetHigh/1$,

cabinetSmall/1, *thingLong*/1 and *thingShort*/1 predicates. The additional requirements of the customer are expressed by:

thingLong(*3*). *thingLong*(*8*).
thingShort(*6*). *thingShort*(*7*). *thingShort*(*4*). *thingShort*(*5*).
thing(*21*). *thingLong*(*21*). *personTOthing*(*1,21*).

The legacy configuration is encoded using *legacyConfig*/1 predicate:

legacyConfig(*cabinet*(*9*)). *legacyConfig*(*cabinet*(*10*)).
legacyConfig(*cabinetTOthing*(*10,8*)).
legacyConfig(*roomTOcabinet*(*16,10*)).
. . .

To implement the configuration requirements of the modified problem we add rules defining the subtypes of cabinets as well as that long things have to be stored in high cabinets. Note, only some of the usual rules for expressing subtypes are needed. Regarding subtypes of thing, no rules are needed at all, because for every *thing* fact either a *thingLong* fact or a *thingShort* fact is contained in the customer requirements and none of these predicates appear in the head of a rule.

1{*cabinetHigh*(*X*),*cabinetSmall*(*X*)}*1* :- *cabinet*(*X*).
cabinetHigh(*C*) :- *thingLong*(*X*), *cabinetTOthing*(*C,X*).

Moreover, each high cabinet requires more space in a room. Such a cabinet occupies two of four slots available in a room, whereas a small cabinet uses only one slot. Note, the last constraint does not allow an answer set where the sum of occupied slots in a room is 5 or more.

cabinetSize(*X,1*) :- *cabinet*(*X*), *cabinetSmall*(*X*).
cabinetSize(*X,2*) :- *cabinet*(*X*), *cabinetHigh*(*X*).
roomTOcabinetSlot(*R,C,S*) :- *roomTOcabinet*(*R,C*), *cabinetSize*(*C,S*).
:- *5*[*roomTOcabinetSlot*(*X,Y,S*) : *cabinetDomain*(*Y*) = *S*], *room*(*X*).

The domains of cabinets and rooms are extended with additional individuals that might be required in a new configuration. The number of new elements in the cabinet and room domains corresponds to the number of things in the modified problem. The upper number *pUpper* of both cabinet and room individuals is set to 7, because 7 things must be stored in the house with respect to the specified customer requirements.

cabinetDomainNew(*22..28*).
cabinetDomain(*X*) :- *cabinetDomainNew*(*X*).
2{*cabinet*(*X*) : *cabinetDomain*(*X*)}*7*.

The modeling of new rooms is done in the same way.

The transformation rules are implemented as described above. For example:

1{*reuse*(*cabinet*(*X*)), *delete*(*cabinet*(*X*))}*1* :- *legacyConfig*(*cabinet*(*X*)).
cabinetDomain(*X*) :- *legacyConfig*(*cabinet*(*X*)).

However, the transformation rules for *legacyConfig*(*person*(*X*)), *legacyConfig*(*thing*(*X*)) as well as *legacyConfig*(*personTOthing*(*X*,*Y*)) could be omitted because facts about persons, things and their relations are given as requirements. Deleting such an atom results in a contradiction.

Given the reconfiguration program, the solver identifies a reconfiguration as well as a set of actions required to transform the legacy configuration into a new one.

For generating optimal reconfigurations we formulate a cost model. The minimization statement in the reconfiguration problem is the same as in the configuration problem. In our reconfiguration example the costs for creation of new high/small cabinets and rooms *cost*(*create*(*a*),*w*) correspond to the costs definition of the configuration problem. To obtain a reconfiguration scenario with the minimal costs of required actions we extend the costs rules described above with costs for creation of new high/small cabinet and room individuals as well as with costs for newly created relations:

cost(*create*(*cabinetHigh*(*X*)),*W*) :- *cabinetHigh*(*X*), *cabinetHighCost*(*W*),
cabinetDomainNew(*X*).

Rules for deducing the costs of reuse and deletion are formulated as described above and the overall reconfiguration costs can be minimized using built-in optimization.

For our example let us assume that the customer sets all deletion costs to 2, whereas reusing has no costs except for cabinets, which could be altered to high in a reconfiguration. The costs of this alteration is set to 3. Creation costs of new high and small cabinets are set to 10 and 5 respectively. Finally, the costs of a new room is set to 5. Creation of relations between individuals is for free. Given these costs assignments, the solver is able to find a set of optimal reconfigurations including the one presented in Figure 3.3.

Modification of the costs results into different optimal reconfigurations. Let us assume the sales-department changes both the costs of deletion of a cabinet and the costs of increasing the height of a

cabinet to 10, and decreases the creation costs of new high and small cabinets to 2 and 1 respectively. In this case the solutions returned by a solver will include the one presented in Figure 3.4.

The suggested knowledge-based method allows to represent a variety of complex (re)configuration problems using answer set programming. In the following chapter we provide the evaluation results of this approach on different (re)configuration problems occurring in practice.

CHAPTER

4 Application cases and evaluation results

In this chapter we illustrate the applicability of our logic-based method for representation and solution to (re)configuration problems. Namely, we evaluate the method presented in Chapter 3 on three different benchmarks: Partner Units Problem (PUP), House problem and Reviewer Assignment Problem (RAP). The first two problems were provided by Siemens who is one of the world-wide leading developers of Customized Configuration Systems for Products and Services [FS14]. The last problem is used to illustrate the broad applicability of our approach, since many real-world problems can be reduced to a configuration problem.

The PUP corresponds to a challenging configuration problem with diverse application domains such as railway safety, security monitoring, electrical engineering, etc. [FHSS11]. Last years the problem has became broad attention for AI in general, it is an important benchmark for logic programming frameworks and it has been thoroughly studied by several researchers [ADG+11, ADF+11, TFF12, Dre12].

The House problem is an abstraction of various configuration problems occurring in practice. One of such problems is the Rack Configuration Problem [MBSF09], which is defined as follows: given a set of hardware modules, a set of racks as well as customer and configuration requirements, the goal is to find such an assignment of modules to racks that all requirements are satisfied. The House problem occurs frequently during configuration of technical systems produced by Siemens. Although the problem reminds of a "toy" example, inclusion of a realistic number of components results in encodings comprising thousands of constraints defined over hundreds of variables. Such large problems are hardly solvable without specifically designed methods [MBSF09, FRF+11a, ADV12, Rya12, RFF13].

The RAP is one of the classical AI problems which has been studied in detail by many researchers, see for example [GS07, FSG+09, GKK+10]. Given a set of papers and a set of reviewers, the problem is to find an assignment of papers to reviewers such that (i) each paper gets at least n competent reviewers and (ii) each reviewer gets at most m papers that fall within her/his competence. In [RPF+12] we show that the RAP corresponds to a configuration problem and show how it can be implemented using Linked Data and ASP. Moreover, we discuss an extension of the original problem for the cases when configuration does not start from scratch, but reconfiguration is required. For instance, if a reviewer drops off and the assigned papers have to be reassigned to some of remaining reviewers.

In the following we describe how knowledge-based configuration and several reconfiguration scenarios can be tackled using ASP systems. We provide the most important evaluation results using general-purpose frameworks and analyze the achieved results at the end of the chapter.

4.1 Problem descriptions

4.1.1 Partner Units Configuration Problem

The Partner Units Problem (PUP) has been recently proposed in [FHSS11] as a challenging problem in automated configuration. It is an abstraction of specific type of configuration problems occurring in practice of Siemens. A sample instance of the PUP can be formulated as follows. A museum has a number of security zones comprising multiple adjoining doors. Each door has a sensor registering when a person enters/leaves the zone. In order to prevent damage to the objects in an exhibition, the number of visitors in a security zone must be restricted. A people counting system configured for a museum comprises a set of control units regulating the number of people in each security zone. If any two security zones with an adjoining door are controlled by different units, then these units must be connected one to each other. The goal is to configure such a system that all door sensors and zones are controlled by a (smallest possible) set of interconnected units.

Formally the problem is defined as follows:

Problem 3 (Partner Units Problem)**.** Let *doorSensor* be a set of door sensors, *zone* be a set of security zones and *unit* be a set of control units. Moreover, let *zone2sensor* be a set of tuples $\langle z, d \rangle$ representing relations between zones and sensors, where $d \in$ *doorSensor* and $z \in$ *zone*. In addition, let *unitCap* and *interUnitCap* denote the maximum number of door sensors/zones and

units, respectively, that can be connected to one unit.

The problem is to find a set $unit' \subseteq unit$ such that assignments of units from the set $unit'$ to door sensors, denoted by *unit2sensor*, zones, denoted by *unit2zone*, and other units, denoted by *partnerunits*, satisfy the following requirements:

- each zone as well as each door sensor must be connected to exactly one unit;
- each unit can control at most *unitCap* door sensors and at most *unitCap* zones;
- if a unit controls a door sensor that contributes to a zone controlled by another unit, then those two units must be connected directly, i.e. one unit becomes a partner unit of the other and vice versa;
- each unit can be connected to at most *interUnitCap* partner units.

The assignments *unit2sensor*, *unit2zone* and *partnerunits* are sets of tuples $\langle u_i, x \rangle$ where $u_i \in unit'$ and x is either a door sensor $d \in doorSensor$, a zone $z \in zone$ or a unit $u_j \in unit'$.

The solution of the problem is *optimal* iff the set $unit'$ is a subset-minimal set of units for which the assignments *unit2sensor*, *unit2zone* and *partnerunits* satisfying all the requirements exist.

Algorithmic complexity of the PUP remains unclear for the problem classes where *interUnitCap* is greater than 2. For the case where $interUnitCap = 2$ the problem is proved to be tractable. The corresponding polynomial-time algorithm is presented in [ADG+11].

In general case, the cardinality of the set $unit'$, i.e. a number of units required to solve the problem, is unknown. Moreover, on the one hand, the set *unit* must comprise enough elements to obtain a solution, i.e. $|unit'| \leq |unit|$. On the other hand, overestimation of the number of required units, i.e. $|unit'| \ll |unit|$, might lead to combinatorial explosion of the search space, making the search for a(n optimal) solution to the PUP infeasible. Therefore, we calculate lower and upper bounds for the number of units required to solve the problem. These bounds provide the minimum and maximum number of elements in the set *unit*, respectively. A lower bound is defined as the number of units required to control all zones and door sensors when all corresponding ports of each unit are used:

$$lower = \left\lceil \frac{max(\#doorSensor, \#zone)}{unitCap} \right\rceil$$

If a solver does not find a solution for a given $lower$ number of units, this number can be iteratively increased. Otherwise an $upper$ bound for the number of units should be provided as well. Since all

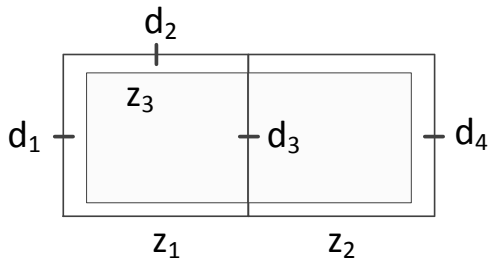

Figure 4.1: The PUP sample instance

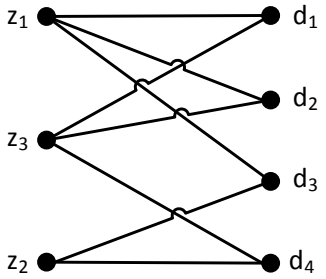

Figure 4.2: The PUP instance as a graph

zones and door sensors must be connected to a unit, we can define an upper bound as follows:

$$upper = \#doorSensor + \#zone$$

Example 1 (PUP Example)**.** Let us illustrate the problem on the following instance presented in Figure 4.1. A security area consists of two rooms that have one adjoining door d_3 and three external doors d_1, d_2 and d_4. The area also has three zones. The zones z_1 and z_2 cover the first and the second room, respectively. Whereas the third zone z_3 covers both rooms. A graph representation shown in Figure 4.2 provides a graphical view of the *zone*2*sensor* relationship of the sample PUP instance. Let each communication unit available to the configurator control at most two zones and two door sensors, i.e. $unitCap = 2$. In addition, let $interUnitCap = 2$. That is, each unit can be connected to at most two other units. Therefore, the lower and upper bounds for the set of units are defined as $lower = 2$ and $upper = 7$. The set of units is generated according to the bounds. Generally, it is safe to generate a set whose cardinality is equal to the upper bound.

Formally, the instance is represented by the following sets:

- $zone = \{z_1, z_2, z_3\}$
- $doorSensor = \{d_1, d_2, d_3, d_4\}$
- $zone2sensor = \{\langle z_1, d_1\rangle, \langle z_1, d_2\rangle, \langle z_1, d_3\rangle, \langle z_2, d_3\rangle, \langle z_2, d_4\rangle, \langle z_3, d_1\rangle, \langle z_3, d_2\rangle, \langle z_3, d_4\rangle\}$
- $unit = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7\}$

One of the solutions of the sample instance, shown in Figure 4.3, is represented by the sets:

- $unit2sensor = \{\langle u_1, d_2\rangle, \langle u_1, d_4\rangle, \langle u_2, d_1\rangle, \langle u_2, d_3\rangle\}$

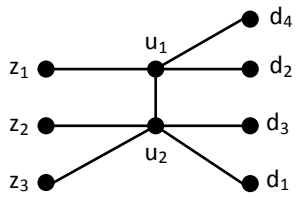

Figure 4.3: An optimal solution of the PUP instance

- $unit2zone = \{\langle u_1, z_1\rangle, \langle u_2, z_2\rangle, \langle u_2, z_3\rangle\}$
- $partnerunits = \{\langle u_1, u_2\rangle\}$

The set of communication units $unit' = \{u_1, u_2\}$ used in the solution comprises 2 elements. This number is the optimal (minimal) number of units for the sample instance and corresponds to the lower bound, defined above. Latter in Section 4.3.1 we describe how the PUP can be modeled and solved using modern approaches. We also evaluate these approaches on a representative set of PUP instances.

4.1.2 Reviewer Assignment Problem

Configuration systems can often be found on the Web. In many application scenarios configuration of products and services follows the standard knowledge-based approach broadly discussed in the literature. However, evolution of the Web technologies pose new challenges for the existing approaches to configuration. Many problems arise, when configuration objects come from an open environment such as third-party websites, or in case of reconfiguration. (Re)configuration is a reasoning task very much ignored in the current (Semantic) Web reasoning literature, despite (i) the increased availability of structured data on the Web, particularly due to movements such as the Semantic Web and Linked Data, (ii) numerous practically relevant tasks, in terms of using Web data, involve (re)configuration. To bridge these gaps, we discuss the challenges and possible approaches for reconfiguration in an open Web environment, based on a practical use case leveraging Linked Data on the Web [HB11] as a "component catalog" for configuration.

The feasibility of (re)configuration based on Open Web Data in a practical scenario can be demonstrated on the Reviewer Assignment Problem (RAP) where the decision if a paper is accepted on a conference depends on reviews made by the program committee. The RAP can be naturally represented as a configuration problem in which reviewers and papers correspond to elements of a

component catalog. A configuration in this case is a set of assignments of papers to reviewers such that (i) the reviewers are interested in reading the paper and (ii) they have enough expertise. These requirements correspond to the requirements of the stable marriage problem, as it is pointed out by Goldsmith and Sloan in [GS07]. A paper/reviewer assignment (marriage) is *stable* if there does not exist an alternative assignment in which paper *and* reviewer are individually better off than in their current assignment. Consequently, a reviewer cannot spot a paper which she/he prefers more and for which she/he has more competence compared to the current assignments.

Preferences for reviewers and papers can be acquired from the conference management systems, such as EasyChair, and Open Web Data. Thus, the preferences of reviewers are provided by reviewers themselves during the bidding process. In our approach we differentiate between four possible bids: (0) *conflict* of interest with the authors of the paper; (1) *indifference*, i.e. no bid is provided; (2) *weak* and (3) *strong* willingness to review the paper. The latter two categories correspond to "I can review" and "I want to review" in the conference system EasyChair. Moreover, we use Open Web Data to derive four categories of expertise of a reviewer w.r.t. a paper: (0) *conflict* if a reviewer is an author of the paper or biased by some other circumstances; (1) *low*, (2) *moderate* and (3) *high* expertise.

Formally, the problem can be defined as follows.

Problem 4 (Reviewer Assignment Problem)**.** Let $reviewer = \{r_1, \ldots, r_n\}$ be a set of reviewers and $paper = \{p_1, \ldots, p_m\}$ be a set of papers. Moreover, let $paperExp = \{\langle p_1, r_1, e_1\rangle, \ldots, \langle p_m, r_n, e_k\rangle\}$ be a set of tuples denoting level of expertise $e = \{0, \ldots, 3\}$ of a reviewer r w.r.t. a paper p and let $reviewerBid = \{\langle r_1, p_1, b_1\rangle, \ldots, \langle r_n, p_m, b_l\rangle\}$ be a set of tuples denoting a bid $b = \{0, \ldots, 3\}$ of a reviewer r for a paper p. In addition, let *rev* denote the required number of reviews per paper and let *minP* and *maxP* denote minimum and maximum number of papers per reviewer, respectively.

The problem is to find a set of assignments $assign = \{\langle r_1, p_1\rangle, \ldots, \langle r_n, p_m\rangle\}$ such that the following requirements are fulfilled:

- each reviewer gets at least *minP* and at most *maxP* papers to review;
- each paper is assigned to *rev* reviewers;
- assignments are non-conflicting, i.e. for any $\langle r_i, p_j\rangle \in assign$ it holds that the bid $b_l > 0$ for $\langle r_i, p_j, b_l\rangle \in reviewerBid$ and expertise $e_k > 0$ for $\langle p_j, r_i, e_k\rangle \in paperExp$;

- number of low quality assignments is minimal, where an assignment $\langle r_i, p_j \rangle$ is of low quality iff (i) $b_l < 3$ for $\langle r_i, p_j, b_l \rangle \in \textit{reviewerBid}$ or (ii) $e_k < 3$ for $\langle p_j, r_i, e_k \rangle \in \textit{paperExp}$;
- the number of unstable assignments is mimimal, where an assignment $\langle r_i, p_j \rangle$ is unstable iff:
  - exists a paper p_k not assigned to the reviewer r_i, i.e. $\langle r_i, p_k \rangle \notin \textit{assign}$, such that $b' > b$ w.r.t. the bids $\langle r_i, p_j, b \rangle, \langle r_i, p_k, b' \rangle \in \textit{reviewerBid}$ and p_k is assigned to a reviewer r_n, i.e. $\langle r_n, p_k \rangle \in \textit{assign}$, for whom the expertise $e' > e$ w.r.t. $\langle p_k, r_i, e' \rangle, \langle p_k, r_n, e \rangle \in \textit{paperExp}$; or
  - exists a reviewer r_n that does not review the paper p_j, i.e. $\langle r_n, p_j \rangle \notin \textit{assign}$, such that $e' > e$ w.r.t. $\langle p_j, r_i, e \rangle, \langle p_j, r_n, e' \rangle \in \textit{paperExp}$ and r_n is assigned a paper p_k, i.e. $\langle r_n, p_k \rangle \in \textit{assign}$, for which $b' > b$ w.r.t. the bids $\langle r_n, p_j, b' \rangle, \langle r_n, p_k, b \rangle \in \textit{reviewerBid}$.

There are several variants of the stable matching which differ from the classic Stable Marriage Problem: *polygamy* – reviewers can get more than one paper and vice versa; *incomplete lists* – some reviewers or papers cannot be assigned to each other; and *indifference* – the preferences express a preset number of preference classes. Each variation of the Stable Marriage Problem mentioned above can be solved in polynomial time [GS07]. The problem becomes more complicated and is known to be NP-hard, if both incomplete lists and indifference occur [MII+02] as in the paper assignment variant of the stable marriage problem. Therefore, a problem solving method which is able to deal with NP-hard problems is required and this justifies the usage of ASP as a problem representation and solving framework.

In order to reduce the load on the solver we consider stability as a soft constraint and minimize the number of assignments which do not fulfill the stability property. In addition, we minimize the number of assignments of papers to reviewers with low and moderate expertise as well as of reviewers to papers with indifference and weak willingness. Our model includes also the following hard constraints: (1) each paper must be assigned to a fixed number of reviewers, (2) assignments must not be conflicting and (3) fairness of the workload should be achieved. In order to distribute the papers among the reviewers as uniformly (fair) as possible, we add a *balancing criterion* as a hard constraint which limits the minimum and maximum number of papers assigned to each reviewer. We use a specific preprocessing step to identify upper and lower bounds by iteratively running a solver. First we set upper and lower bounds equal to $[avg, avg]$, where avg is the average number of papers per reviewer rounded to the next higher integer. We relax subsequently the lower and then

the upper bound until a solution exists. Consequently, we have found boundaries where the variance of the individual workload is as small as possible.

In addition to the stability we can optimize the solution according to the “satisfaction" of papers and reviewers. In particular, the more competent reviewers are assigned to papers the more satisfied are the papers. Conversely, the more reviewer bids are satisfied the more satisfied are the reviewers.

The goal is to assign papers to reviewers with respect to the following optimization criteria which are ordered according to their priority starting with the most important one. The following priorities favor the happiness of papers.

P1 Minimize assignments of papers to reviewers with “low” expertise.

P2 Minimize assignments of papers to reviewers who have only “moderate” expertise on a paper.

P3 Minimize assignments of papers to reviewers who did not provide any preferences for these papers, i.e. “indifference” bid.

P4 Minimize assignments of papers to reviewers who indicated only weak willingness to review them, i.e. “can review” bid.

P5 Minimize the number of assignments which are not stable.

These priorities and any other priority order can be easily encoded within the ASP framework (see Chapter 3).

Consider a small RAP example. Assume there are 6 papers submitted to a conference which program committee includes 6 members, who provided the bids presented in Table 4.1. Each paper should be reviewed three times, therefore 18 reviews are required with an average number of papers per reviewer $avg = 3$. Iterative relaxation of lower and upper bounds allows us to obtain the solution presented in Figure 4.4 with $[avg - 1, avg + 1]$ balancing bounds.

Based on (i) data gathered from `data.semanticweb.org` via SPARQL queries and (ii) the expertise scores computed from this data, we can encode the reviewer configuration problem as an ASP program (following the methodology described in Chapter 3) since ASP is a formalism flexible enough to encode multiple optimization criteria **P1**-**P5**[1].

The output of the ASP program is a configuration of reviewers and papers. However, requirements might change over time demanding changes of a legacy configuration. For instance, in the

[1] The respective SPARQL queries and ASP encodings can be found at `https://sites.google.com/site/reviewersevaluation/`

Papers	Reviewers					
	pc1	pc2	pc3	pc4	pc5	pc6
p1	0/0	1/3	3/2	1/1	1/1	2/1
p2	2/1	2/3	1/0	3/2	1/1	1/1
p3	1/3	0/0	0/0	0/2	2/2	3/1
p4	2/3	2/3	1/2	0/0	0/0	1/1
p5	2/3	1/3	0/0	0/2	1/2	1/1
p6	2/3	0/1	1/0	1/2	1/3	0/0

Table 4.1: Table of paper expertise/reviewer preferences (RAP)

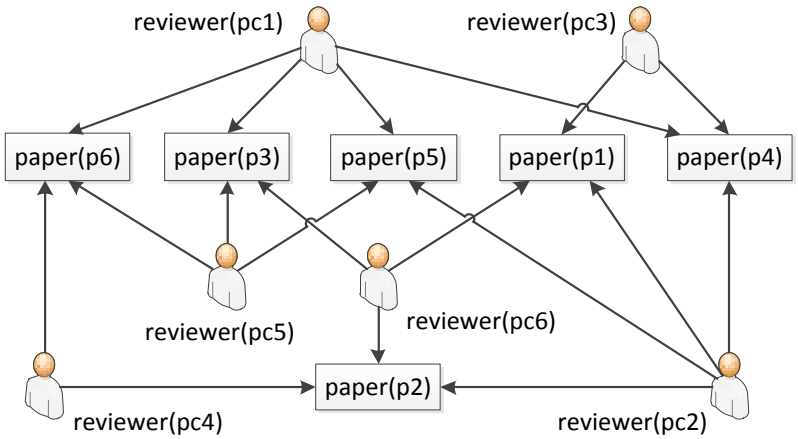

Figure 4.4: RAP configuration solution

Reviewer Assignment Problem, reviewers may drop out, papers could be withdrawn, or additional conflicts of interests may be discovered. Reconfiguration is a transformation of a legacy configuration into a target one such that all current requirements are satisfied. In our application case the legacy configuration is described by the set of papers, reviewers and paper/reviewer assignments. The transformation of the legacy configuration possibly requires that some of its parts are deleted. Therefore, a new configuration problem instance is generated including the current requirements and transformation knowledge regarding reuse or deletion of parts of a legacy configuration. We employ the modeling patterns described in Section 3 to formulate a reconfiguration problem instance. The principle idea is, that for every element of the legacy configuration a decision has to be made whether or not to delete or reuse this element. The reused elements are complemented on demand by addition of new elements in order to fulfill all requirements. For instance, in case a reviewer drops out her assignments to papers must be deleted, eventually new assignments must

be added to other reviewers. If changes happen in the middle of the reviewing process and some papers are already reviewed by PC members, we can declare that these paper/reviewer assignments must be reused for the reconfiguration.

Since we are interested in an optimized reconfiguration solution, costs can be associated to the transformation depending on whether elements are deleted, reused or created. These costs are exploited in an objective function for optimization. For our reconfiguration problem the new sets of papers and reviewers are predefined, so no decision regarding deletion or reuse of these elements has to be made, whereas paper/reviewer assignments may be reused or deleted and possibly new assignments must be created. Consequently, three types of reconfiguration costs can be distinguished:

Creation costs for reviewer to paper assignments absent in the legacy configuration;

Reuse costs for the assignments present in both reconfiguration and legacy configuration;

Deletion costs for the assignments of legacy configuration absent in the reconfiguration.

These domain-specific costs may be defined as needed in order to reflect the preferences of the PC chair. All reconfiguration costs are minimized to obtain the preferred solution. This optimization criterion **PR** extends the set of optimization criteria **P1**-**P5** and possesses the highest priority level. In our application case we assume that both the *reuse* of a paper/reviewer assignment as well as its *deletion* have zero costs; i.e. we assume that reviewers are satisfied if their workload is not changed or even reduced. However, for creating new assignments some costs are associated.

We take the configuration solution from the previous section as a legacy configuration and illustrate some reconfiguration scenarios which may occur in practice:

1. *Late conflict of interest*. Assume the conference submission stage is over, all papers are assigned to reviewers fulfilling the requirements (1) - (3) described previously and a number of reviewers declare late conflicts of interest with papers assigned to them. For example, the program committee member pc5 declares a conflict with the paper p5 assigned to her in a legacy configuration. The reconfiguration process eliminates the existing inconsistent match and assigns p5 to another reviewer. An optimal reconfiguration solution is presented in Figure 4.5.

2. *A reviewer drops out and authors withdraw a paper*. In our example we require reviewer pc3 and paper p1 are excluded from the target configuration. Moreover, all assignments including

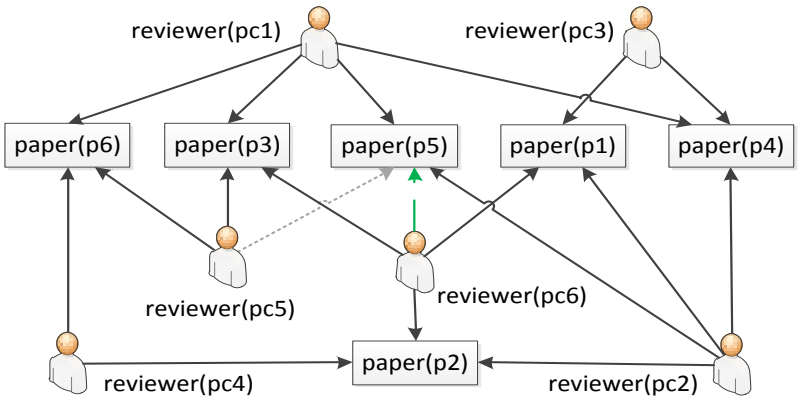

Figure 4.5: RAP reconfiguration solution (scenario 1)

these individuals must be removed as well. All resulting unassigned papers are reassigned to the remaining reviewers in order to fulfill the requirements of the problem. In Figure 4.6 the sample reconfiguration for the mentioned scenario is provided.

3. *Late conflict of interest, a reviewer drops out and authors withdraw a paper.* The last scenario combines the first and the second cases. Reviewer pc5 declares a late conflict of interest with paper p5, reviewer pc3 drops out and authors withdraw their paper p2. An optimal reconfiguration solution is presented in Figure 4.7.

Given their simplicity the reconfiguration solutions of the provided examples are straight forward. However, in the real-world cases presented in the evaluation we observed that complex reassignments are required to fulfill all the requirements, see Section 4.3.2.

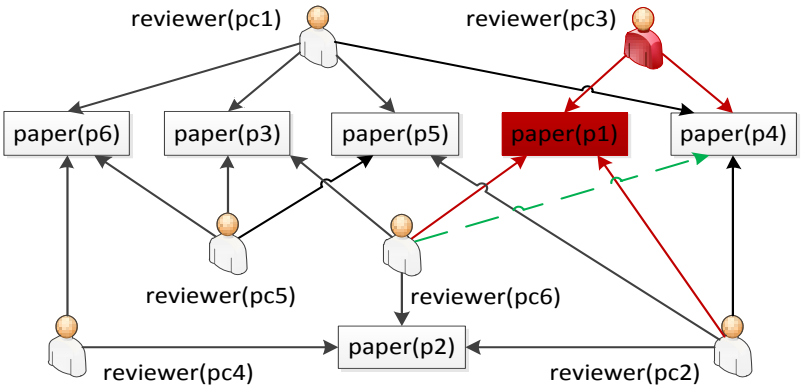

Figure 4.6: RAP reconfiguration solution (scenario 2)

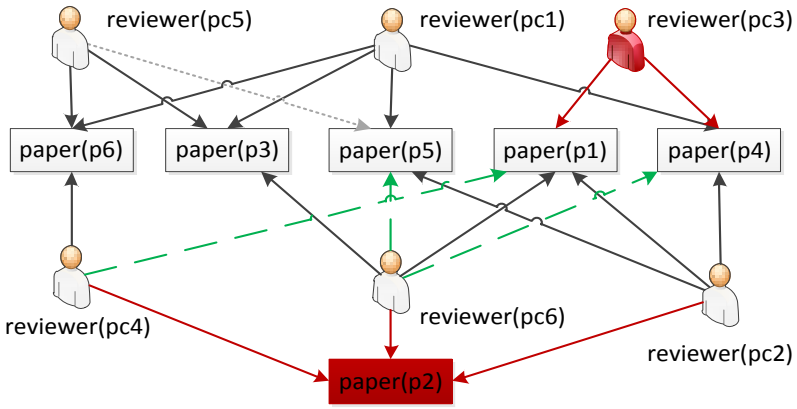

Figure 4.7: RAP reconfiguration solution (scenario 3)

4.1.3 House (re)configuration problem

The last problem we consider is the House (re)configuration problem, House problem for short, which was introduced previously in Section 3.1 (Problem 1 and Problem 2). The House problem is an abstraction of several (re)configuration problems that require selection and assignment of hardware modules depending on the customer and configuration requirements. Such problems frequently occur during configuration of technical systems produced by Siemens, such as telephone switching systems, electronic railway interlocking system, automation systems, etc. Our set of industrial configuration problem instances corresponds to combinatorial problems comprising several optimization criteria. We differentiate between four types of the House (re)configuration scenarios encountered in practice of Siemens:

1. *Empty*: In the empty reconfiguration scenario the legacy configuration is empty and the customer requirements contain sets of things and persons owning 5 things each. Every thing is labeled as short. The reconfiguration process should create missing cabinets, rooms as well as all required relations as shown in Figure 4.8. Actually, the empty instances correspond to configuration problems since no legacy solution is provided.

2. *Long*: The customer requirements of the long scenario specify that each given person owns 15 things. The legacy configuration contains a set of relations that indicate placement of these things into cabinets such that all things of one person are stored in three cabinets that are placed in one room. The customer also requires 5 things of each person to be labeled as long whereas the remaining 10 as short. The goal of the reconfiguration is to find a valid

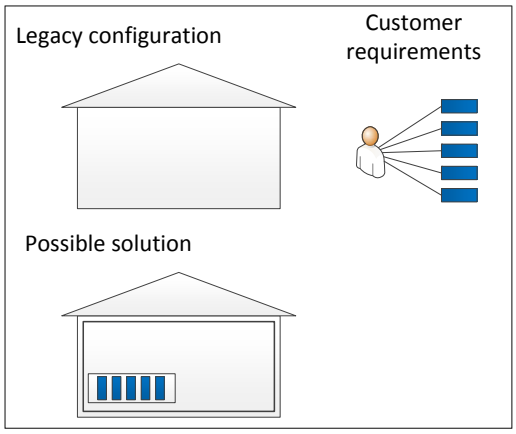

Figure 4.8: House Empty scenario

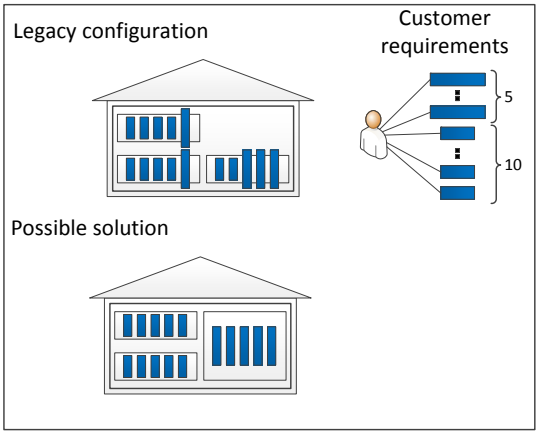

Figure 4.9: House Long scenario

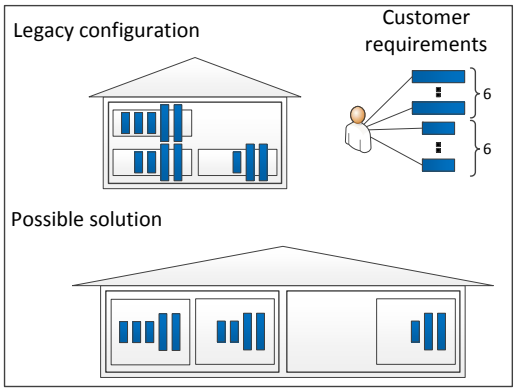

Figure 4.10: House Newroom scenario

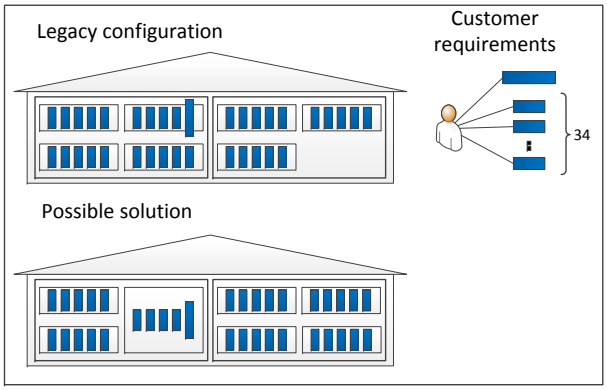

Figure 4.11: House Swap scenario

rearrangement of long things to reused or newly created high cabinets. An instance and its sample solution is presented in Figure 4.9.

3. *Newroom*: The newroom scenario models a situation when new rooms have to be created and some of the cabinets reallocated. In this scenario each person owns 12 things. These things are stored in 3 cabinets placed in one room as indicated by the legacy configuration. In the reconfiguration problem the customer requirements declare 6 of the 12 things as long; and the things can be rearranged as shown in Figure 4.10.

4. *Swap*: The last scenario describes a situation when the customer requirements include only one person, who owns 35 things. In the legacy configuration the things are placed in 3 cabinets in the first room and in 4 cabinets in the second room. Moreover, one of the things in the

second room is labeled as high in the customer requirements. A sample solution corresponds to a rearrangement of the cabinets in the rooms such that a high cabinet can be placed in one of these rooms, see Figure 4.11. The solution depends on the costs schema specified by a customer.

All these scenarios can be easily scaled by increasing the number of things. The number of persons in *Empty*, *Long* and *Newroom* scenarios can always be computed given the number of things. We provide the evaluation results for the realistic problem instances in Section 4.3.3.

4.2 ASP systems and tools

Over decades ASP has drawn attention of many researchers. Their fruitful work resulted in development of numerous ASP solvers. Some of them, such as DLV [LPF+06], SMODELS [SNS02] or NOMORE++ [AGL+05], implement DPLL-like algorithms [DLL62]. While the others, such as ASSAT [LZ04], CMODELS [GLM06] or SAG [LZH06], are build on top of SAT solvers. Finally, a number of solvers including `Clasp` [GKS12] and SMODELS$_{cc}$ implement conflict-driven clause learning algorithms. In the following we present a short overview of the two most actively developed ASP systems, namely, DLV solver and tools from POTASSCO collection which are based on CLASP solver.

4.2.1 DLV

DLV stands for DataLog$^{\vee}$, i.e. Datalog with disjunction, and is a powerful ASP system that is actively developed for more than fifteen years. It is widely used by researchers as a standard reference implementation of ASP paradigm. Recently, this system found also some interesting applications in the industry. The knowledge representation language of DLV system is fully declarative and extends disjunctive ASP programs (see Section 2.3.4) in a number of ways such as: aggregates, weak constraints, functions, lists, etc. Moreover, the system has a number of so called front-ends dealing with specific applications such as planning or diagnosis [EFL+03, EFLP99].

The implementation of DLV is based on solid body of research in databases, optimization methods as well as knowledge representation and reasoning. The computation of an answer set in DLV is split into two phases: grounding and answer set search. In the first phase all variables of the input program are instantiated in a way that the output ground program contains no variables and is semantically equivalent to the input one. In the second phase DLV computes answer sets by applying

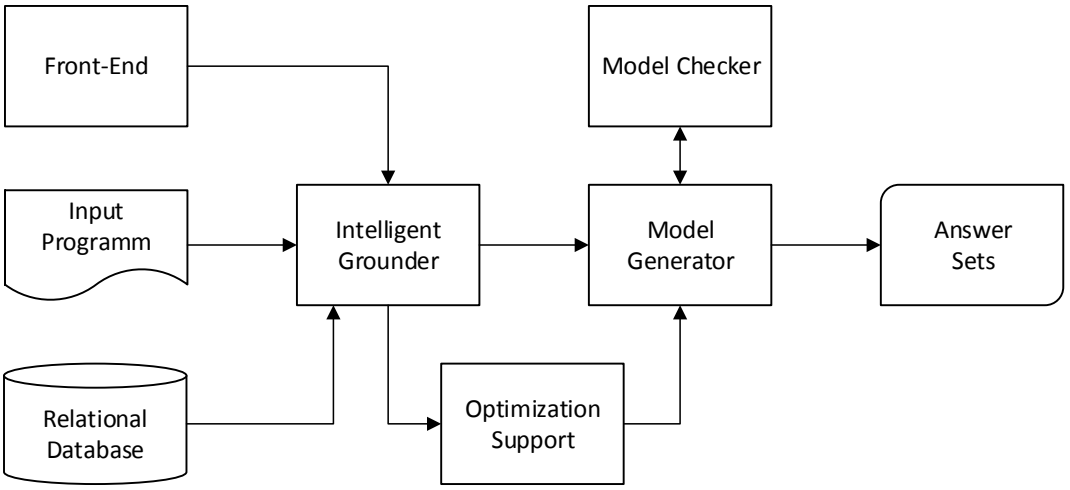

Figure 4.12: Architecture of DLV system

a number of propositional algorithms to the ground program. The general architecture of DLV can be summarized as show in Figure 4.12 [AFL+11].

Intelligent Grounder (IG) [FLP12] is the first component of the system, which takes an input program Π and outputs a subset of a ground program $Gr(\Pi)$ that comprises only those ground rules whose literals can potentially become true. First, IG analyzes the structure of the input program by converting it into a directed graph $G_\Pi = \langle V, E \rangle$, where V is the set of IDB predicates of Π and E contains an arc (a, b) if there is a rule $r \in \Pi$ such that a occurs in the set of head atoms $H(r)$ and b in the set of positive body atoms $B^+(r)$ (see Section 2.3.4). Next, the obtained dependency graph G_Π is analyzed to find strongly connected components which partition the input program into a set of modules $C_1, \ldots, C_n$. The latter are ordered with respect to a precedence relation. This relation ensures that a module C_i precedes C_j, i.e. $i < j$, if instantiation of any atom over a predicate in C_i does not depend on any atom over a predicate in C_j. Such ordering allows incremental grounding which processes one module at a time starting from C_1 up to C_n.

The instantiation procedure takes a module C and a set of atoms S as input, where S is initially equal to EDB of Π. The algorithm outputs a set of ground instances of each rule r such that: (i) there exists an atom $a \in H(r)$ over some predicate in C and (ii) r comprises only atoms from S. Moreover, the set S is updated with all atoms that occur in the head of newly instantiated rules. The

grounding of rules is performed by evaluating the relational join of the positive body literals. IG requires an input program Π to be *safe*, i.e. it ensures that for each variable of the rule $r \in \Pi$ there exists a positive body atom in which it appears. Therefore, the grounding of all literals $B^+(r)$ with respect to the literals in the set S results in the instantiation of all literals in the rule r. After the instantiation, each rule is simplified by removing all positive and negative body literals which are known to be true. Moreover, IG removes all rules whose head is already known to be true or whose body contains a negative literal known to be false.

Answer Set Search approach of DLV is a combination of two algorithms, namely, Model Generator (MG) and Model Checker (MC). The former guesses candidate answer sets and the later verifies whether each guess is an answer set or not. MG implements a backtracking search algorithm which is similar to DPLL procedure [DLL62]. It takes an empty or a partial interpretation I as input and generates a candidate answer set by repeatedly executing the following steps: (1) propagate the deterministic consequences of I (unit propagation) and (2) select an unassigned atom, make an assumption about its truth and add it to I (non-deterministic choice). After each unit propagation step MG checks whether an inconsistency is detected and backtracks if necessary. A completely generated candidate answer set is then provided to MC which checks its stability. In case an answer set is found, the system outputs it to a user, otherwise, MG backtracks to an earlier state of I allowing for generation of an answer set.

Processing of weak constraints is done by Optimization Support (OS) module that acts like a bridge between corresponding parts of IG and MG. In the first step OS guesses the bounds for costs of an optimal solution as well as a sample optimal answer set. Next, it finds all answer sets having that cost. OS module as well as corresponding parts of IG and MG are activated only in presence of weak constraints in the input program.

4.2.2 Potsdam Answer Set Solving Collection

Potassco[2] is a collection of ASP tools which are mainly developed at the University of Potsdam[3]. Among solver Clasp and grounder Gringo the collection comprises a number of tools implementing different extensions of ASP. For instance, Clingcon integrates ASP with Constraint Programming,

[2]http://potassco.sourceforge.net/

[3]In addition, the developers of Potassco maintain Asparagus website, which goal is to gather and make accessible a broad collection of benchmarks for ASP systems: http://asparagus.cs.uni-potsdam.de/

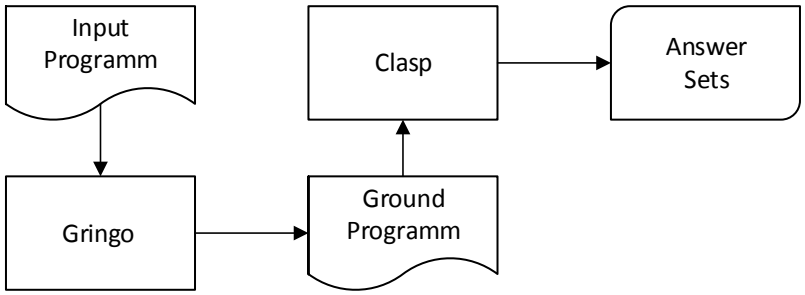

Figure 4.13: Architecture of CLINGO

ASPCUD is an ASP based solver for package dependencies of Debian operation system or COALA is a compiler that transforms programs written in action language such as $\mathcal{C}+$ [GLL+04] to ASP, etc. In the following we present a detailed overview of tools which were evaluated in this work.

Clingo

CLINGO is a versatile ASP solver that won numerous competitions in ASP[4] and related fields such as SAT[5]. The solver comprises two main components depicted in Figure 4.13: GRINGO – an ASP grounder and CLASP – a solver which is applicable to problems formulated in lparse[6] (ASP) or DIMACS[7] (SAT) languages.

The input language of GRINGO extends the normal rules presented in Section 2.3.4 with functions, aggregates, choice rules, optimization statements, etc. [GKKS11]. In addition, it provides a number of interfaces to programming languages such as Lua or Python. The snippets written in these languages allow the grounder to communicate with databases or other programs in order to extend EDB of an input program with required facts. Moreover, this integration often can simplify definition of deterministic computations which can be messy in logic programming. Similarly to Intelligent Grounder (IG) of DLV, GRINGO requires its input programs to be safe, i.e. all rules of the program are safe. In case a safe program does not comprise function symbols of non-zero arity or arithmetic and all rules are safe, it is guaranteed to have an equivalent finite propositional

[4]See, https://www.mat.unical.it/aspcomp2011 or https://www.mat.unical.it/aspcomp2013
[5]http://www.satcompetition.org/
[6]www.tcs.hut.fi/Software/smodels/lparse.ps
[7]ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satreview.ps

representation.

The goal of GRINGO is to generate a succinct propositional representation of an input ASP program. The basic algorithm implemented in GRINGO is based on a lazy evaluation of rules [GKKS12]. Just as IG, this algorithm uses a predicate-rule dependency graph to find strongly connected components. These components are used to partition a logic program into a set of modules which are then grounded in a topological order. Grounding of module rules is done iteratively. Thus, GRINGO finds a subset of rules, which positive body atoms are unifiable with some of the known ground atoms S. The lazy instantiation ensures that in each iteration the grounding algorithm considers only those atoms of S which were instantiated on a previous iteration. This simple technique allows to avoid re-grounding of rules.

CLASP solver is a specifically designed and highly optimized implementation of conflict-driven ASP solving algorithms [GKK+11b]. Some of the techniques were specifically developed for ASP solving and others were adopted from conflict-driven clause learning SAT solvers. The solver takes a ground program $Gr(\Pi)$ as input and starts with an empty or partial interpretation I. The main loop of CLASP can be described as follows:

1. Find a set of atoms S comprising all deterministic consequences of the ground program $Gr(\Pi)$ with respect to the interpretation I and create an extended interpretation $I' := I \cup S$. This computation is specific to ASP and is based on Clark completion [Cla78] as well as detection of unfounded sets [Fab05].

2. If the extended interpretation I' is incomplete, then the algorithm makes a non-deterministic choice by selecting and assigning a truth value to some unassigned atom and continues with the first step.

3. In case an inconsistency (conflict) is detected, the solver backtracks to a point at which the last non-deterministic choice was made. If there is no such point, i.e. a top-level conflict is found, then 'unsatisfiable' is returned. Otherwise, the conflict is analyzed and a corresponding 'conflict constraint' (nogood) is added to the program.

4. If the interpretation I' is complete, then it corresponds to an answer set and is printed to the output.

The model enumeration interface of CLASP implements a novel algorithm that, instead of saving nogoods for each found solution, applies a specific learning/backjumping scheme. This approach

runs in polynomial space and avoids a possible space blowup resulting in the evaluation of programs with large number of solutions [GKNS07]. The enumeration interface handles programs comprising optimization statements in a slightly different way. When a solution is found, the solver computes its costs and generates a specific optimization constraint. The objective function value of this constraint is updated every time a better solution is found. In addition, CLASP uses optimization constraints in its propagation algorithm since these constraints might imply truth values of literals.

Claspfolio

It is well known that modern ASP solvers are highly configurable. That is, there are a lot of parameters (heuristics, clause learning strategies, etc.) which influence the efficiency of the solving process. Selection of appropriate values for the parameters of a solver is very time-consuming, because of the large number of possible parameter-value combinations. Obviously, a straight-forward approach, which tests all parameter-values combinations for a problem instance before starting the search for a solution, is impracticable. Therefore, identification of best settings for a solver has to be done offline, i.e. before the solver is delivered to the users.

The research in this direction resulted in development of portfolio solvers, such as CLASPFOLIO[8]. Development such a solver comprises a training phase which goal is to create a portfolio that associates classes of problems with best-working solver settings (solver configuration). Since classes of problems are unknown, creation of a portfolio is usually based on machine learning techniques. Thus, given a representative set of instances of relevant problems a training algorithm first computes a feature vector for each instance. In modern portfolio solvers a feature vector can comprise over a hundred of values computed by different measures such as problem size, graph statistics, etc. Then, an instance is solved by a solver using different settings and runtime results are associated with the feature vector. Next, a machine learning algorithm is used to find an empirical hardness model, which is a statistical model predicting the runtime of a solver with particular settings for an instance. When a user applies a portfolio solver to find a solution of a previously unseen problem instance, the solver performs the following steps: (i) compute a set of features for the instance; (ii) use empirical hardness model to find the best-working solver settings. The solver is then started with the selected settings.

[8] http://potassco.sourceforge.net/

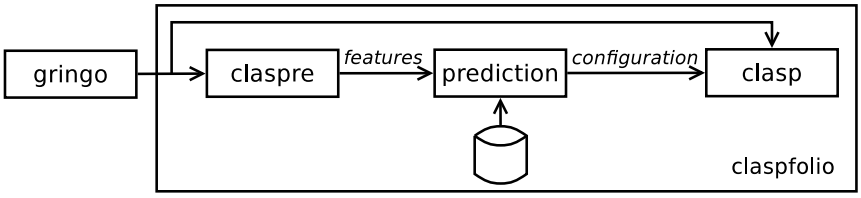

Figure 4.14: Architecture of CLASPFOLIO [GKK+11a]

In our work we applied CLASPFOLIO, which is an ASP portfolio solver. The architecture of CLASPFOLIO [GKK+11a] is provided in Figure 4.14. First, an ASP program is grounded and a light-weight version of CLASP, called CLASPRE, is used to extract features of an instance and even to solve it in case the instance is too simple. If the instance was not solved by CLASPRE, the extracted features are forwarded to the prediction module. The latter uses an empirical hardness model, created by the CLASPFOLIO developers, to find the best solver configuration. Then, this configuration is provided to CLASP which is used to solve the instance. Note that, CLASPRE and CLASP are applied to a copy of the input because the preprocessing done by CLASPRE may decrease the efficiency of settings selected for CLASP by the learnt statistic model (see Figure 4.14).

SBASS: symmetry breaking for ASP programs

Symmetry breaking is a fundamental topic in many AI areas based on combinatorial search. It does not matter which approach we are investigating – CSP, SAT or ASP – elimination (exclusion) of symmetric solutions can speed up the solving process significantly. The pigeon-hole problem is a very good illustrating example. In this problem one has to place n pigeons in m holes such that there is at most one pigeon in each hole. It is clear that there is no sense to distinguish between holes, because all holes are identical. Therefore, all placements of pigeons into holes belong to the same equivalence class of symmetric assignments. This is extremely important if we have $m = n - 1$ holes for n pigeons since one has to verify each of $(n - 1)!$ leaf nodes in a search tree to prove that there is no solution.

In their nature configuration problems are combinatorial (optimization) problems. In order to find a configuration a solver has to instantiate a number of components of some type. In many cases each of these instances can be substituted by another instance in any relation defined for the type. Therefore, similarly to the pigeon-hole problem, many solutions of a configuration problem have

symmetric ones which can be obtained by replacing one component instance with another. These symmetric solutions decrease performance of solving algorithms and have to be pruned out from the search space by symmetry breaking methods.

There are three types of symmetry breaking (SB):

- variable (A, B),
- value $(A, \neg A)$ and
- variable-value $(A, \neg B)$

where A and B are propositional symbols and (A, B) is a permutation that replaces A in all clauses of a CNF (conjunctive normal form) with B and vice versa. While the pigeon-hole problem essentially involves some sort of capacity constraint on a set of interchangeable variables, it exhibits only pure variable symmetries [Sak09]. However, breaking these symmetries improves performance a lot [ARMS03, ASM06, Sak09, DTW11]. Although we are aware of SB research for CSPs, e.g. Gent [GHK02] and Walsh [Wal06], in this section we focus mainly on symmetry breaking approaches for SAT problems. The reason for this is that we want to compare the approach presented in Chapter 3 to the same method but extended with SB predicates.

Modern approaches to identification of symmetries in a CNF are based on the notion of the group isomorphism.

Definition 9 (Group)**.** Group is a structure $\langle G, * \rangle$ where G is a (non-empty) set that is closed under a binary operation $*$ for which the following axioms are satisfied:

- *associativity*: for all $x, y, z \in G,\ (x * y) * z = x * (y * z)$;
- *identity*: there exists an element $e \in G$ such that for all $x \in G,\ x * e = x$;
- *inverse*: for each $x \in G$ there exists $x^{-1} \in G,\ x * x^{-1} = e$.

Note that in the literature the authors often refer G to a group rather than $\langle G, * \rangle$ and omit explicit definition of the operation $*$ and write xy instead of $x * y$.

Let set $G = \{\{A, B\}, \{\neg A, B\}, \{A, \neg B\}, \{\neg A, \neg B\}\}$ include sets of two propositional literals and $* : G \times G \to G$ be a binary operation defined as follows:

	A, B	$\neg A, B$	$A, \neg B$	$\neg A, \neg B$
A, B	A, B	$\neg A, B$	$A, \neg B$	$\neg A, \neg B$
$\neg A, B$	$\neg A, B$	A, B	$\neg A, \neg B$	$A, \neg B$
$A, \neg B$	$A, \neg B$	$\neg A, \neg B$	A, B	$\neg A, B$
$\neg A, \neg B$	$\neg A, \neg B$	$A, \neg B$	$\neg A, B$	A, B

The negation in this operation means that some element of a set should be negated. For instance, in $*(\{\neg A, B\}, \{A, B\})$ the first argument $\{\neg A, B\}$ defines that the first element in a set should be negated and the second argument that none of the elements is negated. In this case the operation negates only A and returns $\{\neg A, B\}$. Clearly, $\langle G, * \rangle$ is a group.

Definition 10 (Subgroup)**.** A group $\langle H, * \rangle$ if a *subgroup* of a group $\langle G, * \rangle$ if $H \subseteq G$ and $H \neq \emptyset$. If $H \subset G$ then H is a *proper subgroup* of G.

Definition 11 (Group generators)**.** Let $H \subset G$ be a subgroup of a group G. The group H *generates* G if all elements of G can be obtained by (multiple) application of the group operation. Elements of H are called *generators* of G. A generator is *redundant* if it can be obtained from other generators. H is *irredundant* if it does not contain redundant generators.

An irredundant generating set of subgroup H provides an extremely compact representation of G. Consider a group $\langle 2\mathbb{Z}, + \rangle$ of all even integers with addition operation. In this case an irredundant set of generators $H = \{-2, 0, 2\}$ provides a group $\langle H, + \rangle$. The notion of generators provides a base for the identification of symmetries in groups. Thus, if a group G contains some subgroup G' which elements can be generated by its subgroup H then we can consider only elements of H. In the context of SB for CNFs the elements of H are symmetry generators. One can use H to declare additional constraints eliminating symmetric solutions.

Another important notion of group theory is group isomorphism. This notion is used to relate different groups, like G' and H from the example given above.

Definition 12 (Group isomorphism)**.** Let $\langle G, * \rangle$ and $\langle G', *' \rangle$ be two groups. Groups $\langle G, * \rangle$ and $\langle G', *' \rangle$ are isomorphic iff there exists an one-to-one (injective) function $\phi : G \to G'$ such that for any two elements $x, y \in G$ and corresponding elements $x', y' \in G'$, i.e. $\phi(x) = x'$ and $\phi(y) = y'$, it holds that

$$\phi(x * y) = \phi(x) *' \phi(y) = x' *' y'$$

That is, if some property is true for the group G it is also true for the group G' and vice versa. Therefore, any group isomorphism maps sets of generators of a group to sets of generators of an isomorphic group.

As we mentioned above symmetric solutions of a configuration problem, and CNFs in general, are obtained by permuting either variables (propositional symbols), their values or both.

Definition 13 (Permutation)**.** A permutation π of a set S is an one-to-one and onto (bijective) function $\pi : S \to S$. Two permutations π and π' can be nested to form a single new permutation function by function composition $\pi''(s) = (\pi \circ \pi')(s) = \pi(\pi'(s))$, where $s \in S$.

It can be shown that the permutation operation is bijective for any given set S and therefore can be used to create a group of permutations.

Definition 14 (Permutation Group)**.** Let A be a non-empty set and S_A be the set of all permutations of A. Then S_A forms a group under permutation operation.

Consider a simple house problem with five persons each owning five things. Given a constraint that restricts placement of things of different persons in the same cabinet, the solution of the problem includes at least five cabinets, e.g. $C = \{1,2,3,4,5\}$. One of the solutions, in this case, will suggest storing all things of the first person in cabinet 1, of the second person in the cabinet 2 and so on. A permutation $(1,2)$ in this context means that the things of the first person will be stored in a cabinet 2 and things of the second person in the cabinet 1. Identification of such permutations in a CNF formula is done through reduction to the colored graph automorphism problem. In order to define this problem let us introduce the group of a colored graph.

Definition 15 (Graph automorphism)**.** Given a graph $GR = (V,E)$ where $V = \{1,2,\ldots,n\}$ is a set of vertices and E is a set of edges. Let $\pi(V) = \{V_1,V_2,\ldots,V_k\}$ be a partition of its vertices, i.e. $\bigcup_{V_i \in \pi(V)} V_i = V$ and $V_i \cap V_j = \emptyset$ for any $V_i,V_j \in \pi(V), V_i \neq V_j$. Then, an automorphism group $Aut(GR,\pi)$ is a the set of permutations of the graph vertices of the same set $V_i \in \pi(V)$ that maps edges to edges and non-edges to non-edges.

To simplify the presentation one can consider $\pi(V)$ as assignment of k different colors to sets of graph vertices. In this case vertices of one color cannot be mapped to vertices of another one. The coloring $\pi(V)$ is *stable* if for all pairs of vertices $u,v \in V$

$$d(u,V_i) = d(v,V_i),\ \forall V_i \in \pi(V) \tag{4.1}$$

where $d(u, V_i)$ is a number of vertices in V_i that are adjacent to u in GR. Given an initial coloring of a graph one can compute a set of different stable colorings. These colorings permutations of the graph vertices correspond to symmetries in the graph and, thus, form a group of permutations.

For the purpose of symmetry breaking in ASP Drescher et al. defined an initial coloring described in [DTW11]. A graph representation of a ground program colored according to this definition can be used as an input to the algorithm computing stable coloring. Given a set of stable colorings it is possible to compute a set of (irredundant) generators. The latter can be used to generate a set of lexicographic constraints that introduce an order on a set of literals – elements of the set of generators. Extension of the ground program with these constraints leads to elimination of symmetric solutions. The meaning of these constraints can be roughly described as: literal b can be in a model only if literal a is.

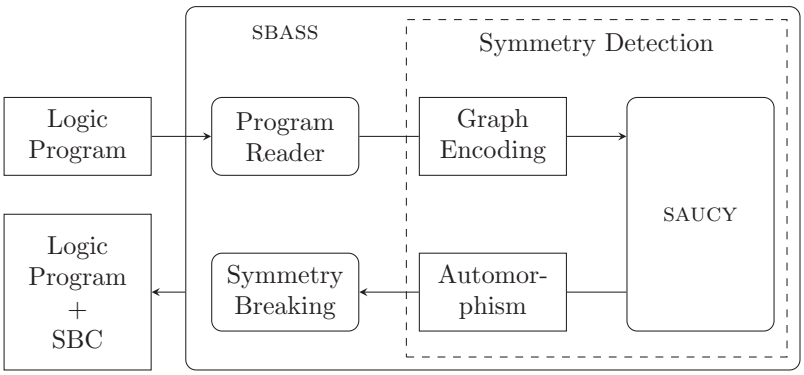

Figure 4.15: Architecture of SBASS [DTW11]

Detection of the equivalence classes for an ASP program can be done by reducing it to the colored graph automorphism problem [DTW11]. In this case, the ground program is represented as a colored graph and the preprocessor SBASS can be used to detect and break symmetries in the search space of ASP instances by adding lexicographic symmetry breaking constraints. The global architecture of SBASS is presented in Figure 4.15 and can be described as follows. SBASS takes a ground logic program produced by the grounder GRINGO as an input. For the given ground program the tool generates a colored graph and provides it as an input to SAUCY[9] [ASM06]. The latter is a graph automorphism identification library that returns a set of graph symmetry generators. Each symmetry generator is used to produce a chain of symmetry breaking constraints (SBCs). The initial

[9]http://vlsicad.eecs.umich.edu/BK/SAUCY/

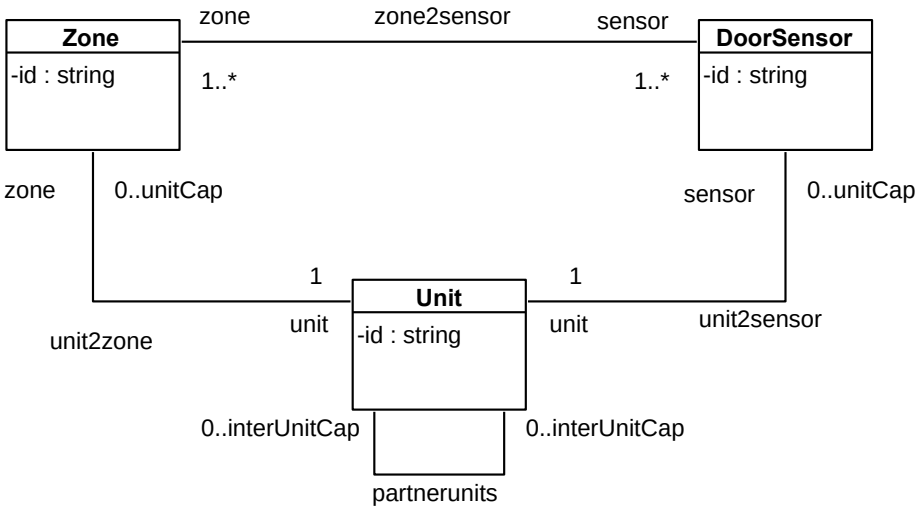

Figure 4.16: UML diagram of the Partner Units Problem

logic program is extended with the SBCs and the result is forwarded to the solver.

SBASS allows to limit the number of computed generators[10], since there are exponentially many generators in the general case [Sak09]. In practice such limitation makes possible computation of SBCs for large ground programs, for which identification of all symmetries would be infeasible.

4.3 Evaluation and analysis

4.3.1 Partner Units Problem experiments and discussion

Solutions to the PUP, which is a typical configuration problem, can be obtained in different ways. Most of the approaches presented in Section 2.3 are suitable for this task. For instance, the UML representation of the PUP, depicted in Figure 4.16, shows that the problem comprises 3 classes: *Zone*, *DoorSensor* and *Unit* representing the set of zones, door sensors and units, respectively. The associations between the classes encode the input relation *zone*2*sensor* as well as output relations *unit*2*sensor*, *unit*2*zone* and *partnerunits*. The cardinalities of the associations capture requirements for assignments of zones and door sensors to communication units as well as connections between units.

[10]command line option `-limit=n`

The ASP-based knowledge representation language for configuration problems described in Section 3 can also be used to represent the PUP in a very concise and elegant manner. The elements of input sets of zones, door sensors and units are mapped to atoms over *zone*/1, *doorSensor*/1 and *unit* /1 predicates. Each tuple of the input relation between zones and sensors is mapped to atoms over the binary predicate *zone*2*sensor*/2. The output schema of a PUP encoding comprises predicates *unit*2*zone*/2, *unit*2*sensor*/2 and *partnerunits*/2. All atoms over predicates of the output schema in an answer set of a program encode a configuration, i.e. a solution of the PUP. The set *unit′* of communication units used in the solution can be obtained by extraction of first terms of atoms over *unit*2*zone*/2 and *unit*2*sensor*/2 predicates.

Example 2 (PUP example, cont.)**.** In the ASP representation the input sets describing the sample PUP instance can be encoded by the facts (lines 1-5), given in Listing 4.1. In addition to facts, the program comprises four constants providing the lower and upper bounds for the set of units (line 6) as well as their capacities (line 7). These constants can also be declared by means of facts. Note that, in our approach the set *unit* of communication units is not a part of the input, but is generated in a program according to the bounds and a solution strategy (heuristics). This allows to increase the efficiency of solving in many practical cases.

```
1 doorSensor(d1). doorSensor(d2). doorSensor(d3). doorSensor(d4).
2 zone(z1). zone(z2). zone(z3).
3 zone2sensor(z1,d1). zone2sensor(z1,d2). zone2sensor(z1,d3).
4 zone2sensor(z2,d3). zone2sensor(z2,d4).
5 zone2sensor(z3,d1). zone2sensor(z3,d2). zone2sensor(z3,d4).
6 #const lower = 2. #const upper = 7.
7 #const unitCap = 2. #const interUnitCap = 2.
```

Listing 4.1: Facts encoding the sample PUP instance

A number of ASP solvers can be used to find answer sets which correspond to possible configurations. Two ASP systems were examined in this work: DLV, i.e. disjunctive datalog[11] as well as the Potsdam Answer Set Solving Collection (Potassco)[12].

[11] http://www.mat.unical.it/dlv-complex

[12] http://potassco.sourceforge.net/

Each Partner Units Problem instance can be represented as:

1. an extensional database (EDB), which contains all input facts, e.g. as in Listing 4.1, and

2. an intentional database (IDB or just program) including rules and constraints.

Separating a problem instance into two parts helps to simplify both the understanding and implementation of the program, since one program can be used for multiple EDBs. Moreover, one EDB describing the Partner Units Problem instance can be used with programs written for different solvers. Note that an EDB and an IDB correspond to the customer and configuration requirements respectively as consistent with the definitions given in Section 2.2.

In the remainder of this section we provide modeling patterns for the PUP. For each pattern we discuss some of the evaluation results on the benchmarks we submitted to the Third ASP Competition 2011[13] [CIR14]. The test instances were generated by Siemens and can be divided into two sets: Partner Units Polynomial (*interUnitCap* = 2) and Partner Units (*interUnitCap* > 2). The numbers of zones and door sensors in an instance vary from 20 to 298 and the number of units from 14 to 149 with 50 units on average. The instances are structured as follows:

- small-* are of a small size (up to 9 zones/door sensors) without any specific structure;

- double-* consist of two rows of zones with all interior doors equipped with a sensor;

- double_v-* are the same as double*, except that there are additional zones covering other zones;

- triple-* are grids with only some of the doors equipped with sensors and there are additional zones that cover multiple zones;

- grid-* are not full grids, but some doors are missing, and there are no zones without doors.

Later in this section, we will present evaluation results using our best found ASP encoding/solvers and compare it to the results obtained by means of constraint programming, propositional satisfiability testing and integer programming systems [ADF+11].

[13] https://www.mat.unical.it/aspcomp2011/OfficialProblemSuite

DLV representation

The DLV encoding of the PUP is presented in Listing 4.2. The *unit2zone*, *unit2sensor*, and *partnerunits* relations are encoded using *unit*2*zone*/2, *unit*2*sensor*/2 and *partnerunits*/2 predicates, as defined in Section 4.1.1. The first rule (line 1) of the encoding generates number of units corresponding to the upper bound *upper*. Then, the first block of rules consists of a disjunctive rule[14] and two constrains. This modeling approach corresponds to Guess/Check/Optimize (GCO) paradigm presented in [LPF+06], where the disjunctive rules are used to define the search space of a problem. In case of PUP, the answer sets of the program comprising only a disjunctive rule (line 3) represent all possible connections between zones and units. Of course, this rule allows undesired situations. For instance, if an answer set comprises only negated atoms over *unit*2*zone*/2 predicate, then it corresponds to a solution candidate in which neither zones nor door sensors are connected to any units. This candidate definitely violates the first requirement of the problem. To exclude such "invalid" solution candidates we introduce the Check part of the program comprising two integrity constraints. These constraints (lines 3 and 4) require each answer set to comprise sets of atoms representing that at least 1 and at most *unitCap* zones are connected to a unit. Similarly, the *unit*2*sensor* relation is encoded in the second block of the program (lines 5-7). Finally, the third block defines the *partnerunits* relation (lines 8 and 9). The integrity constraint in line 10 specifies that each unit can communicate with at most *interUnitCap* partner units.

An optimal solution, i.e. with the minimal number of units, can be found using the optimization service of DLV. The idea is to add optimization statements, called weak constraints, to a program that allow to find a solution with minimal number of units. A weak constraint is an expression of the form:

$$:\sim\ l_1, \ldots, l_k,\ not\ l_{k+1}, \ldots,\ not\ l_m.\ [weight : level]$$

where for $0 \leq i \leq m$ l_i are literals, *weight* and *level* are non-negative integers denoting weight and priority level of a weak constraint. Given a program with weak constraints DLV returns answer sets, which minimize the sum of weights of violated weak constraints in priority level starting from the highest one. That is, the solver finds answer sets with minimal sum of weights on the highest level. Then, among those answer sets it selects the ones with minimal sum of weights on the second highest level, and so forth.

[14] "v" stands for a disjunction

```
1  unit(1 . . upper).

2  unit2zone(U,Z) v −unit2zone(U,Z) :- zone(Z), unit(U).
3  :- unit(U), not #count{Z : unit2zone(U,Z)} <= unitCap.
4  :- zone(Z), not #count{U : unit2zone(U,Z)} = 1.

5  unit2sensor(U,D) v −unit2sensor(U,D) :- doorSensor(D), unit(U).
6  :- unit(U), not #count{D : unit2sensor(U,D)} <= unitCap.
7  :- doorSensor(D), not #count{U : unit2sensor(U,D)} = 1.

8  partnerunits(U,P) :- unit2zone(U,Z), unit2sensor(P,D), zone2sensor(Z,D), U != P.
9  partnerunits(U,P) :- partnerunits(P,U).
10 :- unit(U), not #count{P : partnerunits(U,P)} <= interUnitCap.
```

Listing 4.2: DLV encoding of the PUP

```
1  unitUsed(U) :- unit2zone(U,Z).
2  unitUsed(U) :- unit2sensor(U,D).
3  :∼ unitUsed(U). [1 : 1]
4  :- #count{X : unitUsed(X)} < lower.
5  unitUsed(X) :- unit(X), unit(Y), unitUsed(Y), X < Y.
```

Listing 4.3: Optimization rules for the DLV encoding of the PUP

To minimize the number of units we extend the program given in Listing 4.2 with the rules presented in Listing 4.3. The first two rules are used to derive atoms over a *unitUsed*/1 predicate every time a unit is connected to a zone (line 1) or to a door sensor (line 2). Next, we define a weak constraint (line 3) that is violated by every atom over the *unitUsed*/1 predicate which is true in an answer set. The less such atoms are in an answer set the lower is the sum of weights associated with this answer set. Consequently, DLV will return only answer sets corresponding to PUP solutions comprising the minimal number of units. Computation of a solution with optimal number of units can be speed up by inclusion of the two rules: (i) a constraint (line 4) which requires the number of used units to be greater or equal to the lower bound. And (ii) the ordering rule (line 5) which filter outs symmetric optimal solutions resulting in renaming of the units.

We evaluated the program presented in Listing 4.2 using the PUP benchmarks described above.

```
1 unit(1 . . lower).

2 unit2zone(1,Z) v unit2zone(2,Z) v . . . v unit2zone(lower,Z) :- zone(Z).
3 unit2sensor(1,D) v unit2sensor(2,D) v . . . v unit2zone(lower,Z) :- doorSensor(D).
4 :- unit(U), not #count{Z : unit2zone(U,Z)} <= unitCap.
5 :- unit(U), not #count{D : unit2sensor(U,D)} <= unitCap

6 partnerunits(U,P) :- unit2zone(U,Z), unit2sensor(P,D), zone2sensor(Z,D), U != P.
7 partnerunits(U,P) :- partnerunits(P,U).
8 :- unit(U), not #count{P : partnerunits(U,P)} <= interUnitCap.
```

Listing 4.4: Improved DLV encoding of the PUP

All small instances were solved by the DLV solver within the time frame of 180 seconds (see Section 4.3.1). However, we observed the significant reduction of performance for mid-sized instances. For example, for an instance including 20 zones the computation time was greater allowed 180 seconds in all tests. For the large instances the grounding process of the DLV solver (version 4.6.1) was not finished within the runtime limit. Namely, the grounder failed to process instances with more than 58 units.

Therefore, the program was improved as shown in Listing 4.4. Namely, we model the *unit* 2*zone* relation as follows: a zone is controlled by the first unit, or the second, or the third, etc. (line 2). The same modeling approach was used to represent the *unit*2*sensor* relation (line 3). The *partnerunits* relation representation remains the same as in the ASP program given in Listing 4.2. The improved encoding cannot be formulated in the DLV language completely, since the language does not support parametric connectives. Therefore, the program must be generated for each EDB by an external tool. Given the improved program the DLV grounder was able to ground all the instances of our benchmarks. Also the solver computed solutions faster.

Potassco representation

In addition to DLV, we implemented the Partner Units Problem using Clingo solver from Potassco toolkit. This solver integrates the grounder Gringo and the solver Clasp in a monolithic way. In the

GRINGO knowledge representation language parametric connectives can be modeled using cardinality constraints as heads of rules. That is, rules of the form:

$$l\ \{l_1,...,l_k\}\ u\ \text{:-}\ l_{k+1},\ldots,l_m,\ not\ l_{m+1},\ldots,\ not\ l_n.$$

where l_i for $0 \leq i \leq n$ are literals and the bounds $0 \leq l \leq u$ are non-negative integers (see Section 3.3). Each cardinality constraint in the head of such rule specifies that at least l but at most u ground literals of the set $\{l_1,...,l_k\}$ must be true. The sets of ground literals $\{l_1,...,l_k\}$ can be generated by the grounder automatically from conditional expressions of the form:

$$l\ :\ l_1\ :\ \ldots\ :\ l_k$$

where l and l_i for $0 \leq i \leq k$ are literals. The symbol ":" restricts the instantiation of variables to those constants that satisfy the condition. For example, given 2 facts $unit(1)$. and $unit(2)$. the conditional literal in the cardinality constraint 1 $\{unit2zone(U,z)\ :\ unit(U)\}$ 1. expands to the cardinality constraint 1 $\{unit2zone(1,z),\ unit2zone(2,z)\}$ 1.

The program presented in Listing 4.5 is similar to the DLV implementation in Listing 4.4 up to the Guess part (lines 2 and 4). Thus, the first rule (line 1) generates the required number of units represented as facts: $unit(1)$. $unit(2)$. ... $unit(upper)$. However, instead of disjunctive rules this encoding uses cardinality constraints in the heads of the second and the fourth rules. These constraints ensure that each zone and each door sensor is connected to exactly one unit. The third and the fifth rules are constraints that guarantee each unit to control at most *unitCap* zones and *unitCap* sensors. The last three rules define the connections between units and limit the number of partner units to *interUnitCap*. Note that rules 3, 5 and 8 can be rephrased by moving the cardinality constraint on the left-hand-side of the rule and adapting the boundaries. We used the depicted encoding, because it follows exactly the GCO paradigm [LPF+06]. As our experiments showed, depending on the particular encoding runtimes may slightly vary (see Section 4.3.1).

As we already mentioned in Section 4.1.1, in general, we do not know how many units are required to solve the problem. Therefore, in Listing 4.5 we generate an *upper* number of units required in the worst case to find a solution. Just as in case of DLV, the optimal solutions comprising the minimal number of units can be computed using CLINGO built-in optimization services. The encoding of optimization problem, given is Listing 4.6, is similar to the DLV encoding presented in

```
1 unit(1 .. upper).

2 1{unit2zone(U,Z) : unit(U)}1 :- zone(Z).
3 :- unit(U), unitCap + 1{unit2zone(U,Z) : zone(Z)}.

4 1{unit2sensor(U,D) : unit(U)}1 :- doorSensor(D).
5 :- unit(U), unitCap + 1{unit2sensor(U,D) : doorSensor(D)}.

6 partnerunits(U,P) :- unit2zone(U,Z), zone2sensor(Z,D), unit2sensor(P,D), U != P.
7 partnerunits(U,P) :- partnerunits(P,U), unit(U), unit(P).
8 :- unit(U), interUnitCap + 1{partnerunits(U,P) : unit(P)}.
```

Listing 4.5: Initial Clingo encoding of the PUP

```
1 unitUsed(U) :- unit2zone(U,Z).
2 unitUsed(U) :- unit2sensor(U,D).
3 #minimize{unitUsed(X)}.
4 lower {unitUsed(X) : unit(X)} upper.
5 unitUsed(X) :- unit(X), unit(Y), unitUsed(Y), X < Y.
```

Listing 4.6: Finding a solution with optimal number of units

Listing 4.3 and differs only in rules 3 and 4. Since, weak constraints are not supported by Clingo, we have to use special optimization atoms of the form:

$$\#minimize\,\{l_1 = w_1@p_1, \ldots, l_n = w_n@p_n\}\,.$$

where l_i, w_i and p_i for $0 \leq i \leq n$ is a literal, a weight and a priority, respectively. Weights and priorities are integers and are treated by the optimization in the same way as the weights and priority levels of weak constraints in DLV. By default, the values of all weights and priorities are set to 1. Therefore, the rule 3 requires the solver Clasp to find an answer set that comprises a minimal set of atoms over *unitUsed*/1 predicate. The cardinality constraint in rule 4 required that at least *lower* and at most *upper* atoms over *unitUsed*/1 predicate must be true in every answer set.

The evaluation of Potassco implementation given in Section 4.3.1 showed its efficiency for mid-size as well as for many large problem instances provided by Siemens. Namely, it solved

Test case	Input	interUnitCap=4		interUnitCap=3		interUnitCap=2	
		Clasp	Claspfolio	Clasp	Claspfolio	Clasp	Claspfolio
double-20	20Z,28S,14U	0:00.07	0:00.07	0:00.08	0:00.10	0:04.33	0:00.68
double-40	40Z,58S, 29U	0:02.29	0:00.72	1:50.67	0:05.18	3:42.38	5:40.10
double-60	60Z,88S, 44U	1:55.33	0:05.60	37:34.89	22:21.08	timeout	timeout
double-80	80Z,118S,59U	15:41.67	6:44.38	timeout	timeout	timeout	timeout

Table 4.2: Selected PUP results for the double-* instances

all instances except double-* instances of medium and large size. Therefore, we consider only the Potassco encodings, such as presented in Listing 4.5, for the PUP and other application cases represented in this chapter.

By the application of Claspfolio descibed in Section 4.2.2 we identified the most promising options for the PUP test instances and ran the benchmarks using them. The following parameters turned to be the most effective for the PUP test instances and the program presented in Listing 4.5:

- *heu*=VSIDS, i.e. Variable State Independent Decaying Sum heuristic;
- *del*=3,1.1,1000 which fixes the size and growth factor of the dynamic nogood database;
- *restarts*=100,1.5,20000 which parametrizes a restart policy;
- *local-restarts* which exploits local restarts.

Table 4.2 shows the selected results for the double-* instances. The first and the second columns include name of an instance and the number of zones/door sensors/units in a configuration solution. We ran instances with a different value for *interUnitCap*: 2, 3 and 4 for one hour and recoded the runtimes using the default options of Clasp, denoted **Clasp** in Table 4.2, and using the suggested by Claspfolio parameters, denoted **Claspfolio**. The runtimes are given in minutes. Our results can be summarized as follows. The usage of learning techniques, the application of VSIDS heuristic [MMZ+01] instead of the default BerkMin heuristic [GN07] in Clasp as well as the exploitation of local restarts [RS08] allow to speed up the computations. Namely, Clasp performs better for the most of the double-* instances and for other types of instances, but does not deliver a solution in 60-minute time frame for the mid-sized and large double-* instances.

Incorporation of domain-specific heuristics

The fact that there should exist a polynomial-time algorithm for *interUnitCap* = 2 mentioned in [ADG+11] and bad performance of the solver for the double-* instances forced us to exploit domain-specific knowledge about the problem. In many cases this knowledge makes possible development of heuristics which allow to solve practical problems in an admissible time. We analyzed the structure of the PUP instances and created 6 different ASP implementations which variously restrict either zones or door sensors assignments to units, but preserve all solutions. The evaluation results for these methods are quite unstable, that is, there was no program which computed solutions equally good for a class of instances. Therefore, we omit the whole description of these heuristics and only give a short description of the best two from them, called *ring* and *pearl* heuristics.

The *ring* heuristic was initially suggested in [ADG+11] and is based on the observation that the unit graph in a solution of the connected PUP instance is always either a path or a cycle. In other words, a ring structure of the problem can be exploited. The authors developed an algorithm, called DECPUP, which showed to be very efficient on the double-* instances with *interUnitCap* = 2, i.e. the hardest instance with 200 zones and 298 door sensors can be solved in about 1 second time.

We expressed the same heuristic in ASP using the following rules which we added to the program given in Listing 4.5:

```
1 partnerunits(U1,U2) :- unit(U1), unit(U2), U1 = U2 - 1.
2 partnerunits(lower,1).
```

Listing 4.7: “Ring” heuristic

This simple extension of the program allows to finish the solving process in an order of magnitude faster for the double-* and other instances with *interUnitCap* = 2. However, it did not outperform the DECPUP algorithm on the double-* instances.

Therefore, we developed so called *pearl* heuristic for the double-* instances. This is an ASP program which extends program given in Listing 4.5 by a set of positive normal rules, i.e. the rules of the form

$$a \text{ :- } l_1, \ldots, l_n.$$

where a and l_i for $0 \leq i \leq n$ are literals. These rules sequentially assign adjacent zones and door

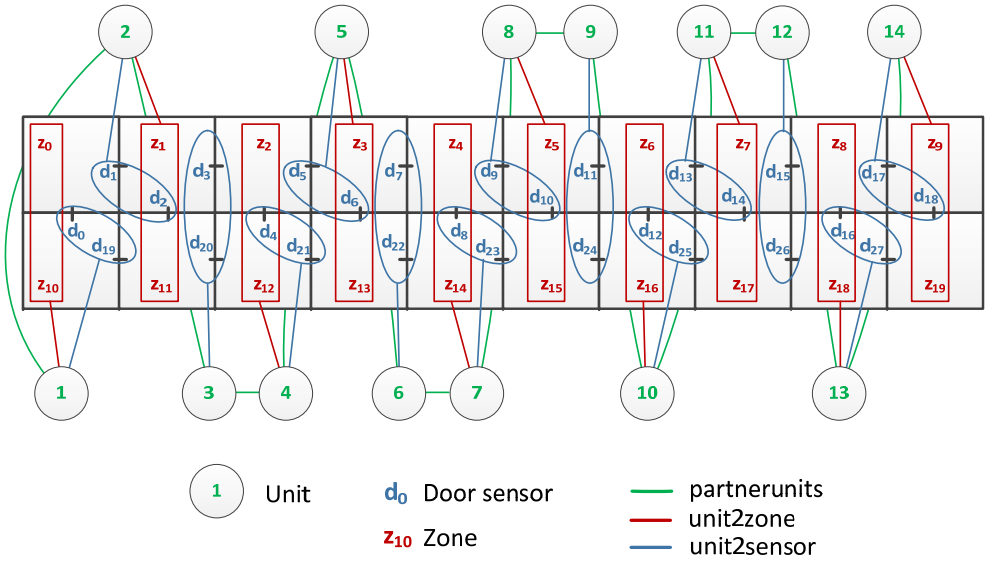

Figure 4.17: "Pearl heuristic" for the double-* test cases

sensors to units in the predefined order shown in Figure 4.17 (see the program in Appendix A). It allows to solve all double-* instances efficiently exploiting a general-purpose ASP solver CLINGO, e.g. double-80 can be solved in 5 seconds. For comparison, application of CLINGO to the program given in Listing 4.5 without the suggested ordering did not return a solution after 100 minutes. This means that, additional knowledge about the inputs can be exploited to achieve significant improvements. Nevertheless, the hardest double-* instance can be solved in about 1 minute time using the *pearl* heuristic, expressed using ASP rules, which is approximately 60 times worse than using the DECPUP algorithm presented in [ADG+11]. Moreover, it turned to be hard to generalize the *pearl* heuristic for all other test cases with $interUnitCap = 2$.

However, it is worth to mention that we have not done much research in finding of efficient heuristic methods for the PUP, since the main topic of this work was the development of a general knowledge-representation language for (re)configuration problems. The more sophisticated and powerful methods for the Partner Units Problem were elaborated by Teppan et al. in [TFF12] and Drescher in [Dre12].

Discussion and related work

The results for the PUP can be summarized as follows. ASP languages such as DLV and Potassco are concise and expressive enough to capture configuration knowledge. The performance of a configurator depends on a particular encoding of a problem, a chosen solver as well as its parameters. Generally, we achieved the best performance using Potassco implementation given in Listing 4.5 and Claspfolio solver.

Description of the problem as well as our encodings were submitted to the Third ASP competition 2011. The goal was to introduce the PUP to a broad consideration in ASP community taking part in two competition tracks: System Track and Model&Solve Track. In the System Track competition participants are not allowed to change the encoding of the problem and run their solvers on a given set of benchmarks. Due to the requirements of the competition, the set of benchmarks was limited to the instances for which we could provide an evidence that they can be solved in an admissible time. In the Model&Solve Track the participants can introduce a competitive encoding of the problem.

The Partner Units Polynomial Problem participated in the System track where 11 ASP solvers contested. Similar to our results, Potassco solver Clasp showed the best performance[15]. Both problems, Partner Units and Partner Units Polynomial took part in the Model&Solve Track where 6 ASP teams competed. Potassco team outperformed other teams on our benchmarks[16]. They developed an encoding which was equivalent to ours, except that the ring heuristic was not exploited, and used Clasp and aClasp with parameters of the solver which were a bit different to ours. Therefore, our results are slightly better on average than the results obtained in the competition.

We have compared different ASP representations/solvers so far. To make a proper evaluation other general-purpose frameworks have to be investigated. In [ADF+11] Aschinger et al. presented the evaluation results which compare encodings for the PUP using state-of-the-art propositional satisfiability (SAT), mixed integer programming (MIP), constraint solving (CSP) and answer set programming (ASP). We do not present details regarding the corresponding encodings, they can be found in [ADF+11], but only analyze the evaluation results below [ADF+11].

The set of benchmarks received from Siemens described at the beginning of this section was evaluated using the following approaches:

[15] https://www.mat.unical.it/aspcomp2011/SystemTrackFinalResults

[16] https://www.mat.unical.it/aspcomp2011/Model%26SolveTrackFinalResults

- Constraint programming: ECLiPSe-Prolog (v. 6.0)[17] (CSP);
- Propositional satisfiability testing: MiniSat (v. 2.0)[18] (SAT);
- Polynomial algorithm implemented in Java (DECPUP);
- Answer set programming: Clingo (v. 3.0) and Claspfolio (ASP);
- Integer programming: Cbc (v. 2.6.2)[19] in combination with Clp (v. 1.13.2) (CBC) and Cplex (v. 12.1)[20] (CPLEX).

In the ASP, SAT and CSP models, as well as in DECPUP, we use iterative deepening search for finding optimal solutions, as this has proven to be the more efficient than to use the built-in optimization. The optimization was not exploited in the integer programming model since the objective function worsened performance. Both SAT and MIP solvers were used out of the box, whereas for the CSP and ASP model the ring heuristic outlined in Section 4.3.1 was exploited if $interUnitCap = 2$. Additionally, if $interUnitCap > 2$ for the ASP model we employ the solvers parameters identified by Claspfolio. It is likely that the similar machine learning techniques could also fruitfully applied in the other frameworks, but they were not developed.

The runtimes obtained for these problem encodings are shown in Table 4.3 and in Table 4.4[21]. The first column in both tables represents a name of an instance. The next three columns include the number of zones/door sensors/units contained in a configuration. The symbol '*' indicates a timeout and the symbol '/' denotes that the problem instance has no solution, i.e. the solver has to find that the instance is unsatisfiable. The runtime for each approach is given in seconds. Note that we omit the results for the small-* instances due to their simplicity.

Let us first analyze the results presented in Table 4.3 which were obtained for the PUP if $interUnitCap = unitCap = 2$. The combination of assuming a fixed cyclic unit graph (ring heuristic) resulted in drastic speed-ups for the CSP, SAT and ASP solvers. The ASP and SAT encodings show quite similar behavior since both Clingo and MiniSat use variations of the DPLL-procedure [DP60] for reasoning. They even both failed to solve all double instances and get faster at some point as

[17] http://eclipseclp.org/
[18] http://www.minisat.se/
[19] http://www.coin-or.org/
[20] http://www.ibm.com/
[21] All experiments were performed on a 3 GHz dual core machine using 4 GB RAM running Fedora Linux, release 13 (Coddard) and using a 10-minute time frame (600 seconds).

Name	Zones	Sensors	Units	CSP	SAT	DECPUP	ASP	CBC	CPLEX
double-20	20	28	14	0.02	0.48	0.01	0.16	14.12	1.53
double-40	40	58	29	0.28	2.36	0.05	3.93	224.14	13.58
double-60	60	88	44	0.42	29.74	0.08	*	*	213.58
double-80	80	118	59	1.14	*	0.16	*	*	522.50
double-100	100	148	74	1.89	*	0.41	*	*	*
double-120	120	178	89	3.21	*	0.39	*	*	*
double-140	140	208	104	5.01	*	0.59	*	*	*
double-160	160	238	119	13.94	*	0.71	*	*	*
double-180	180	268	134	20.07	*	0.87	*	*	*
double-200	200	298	149	14.4	*	1.08	*	*	*
double_v-30	30	28	15	0.09	0.42	65.49	0.26	37.18	2.93
double_v-60	60	58	30	0.26	3.15	*	1.94	*	*
double_v-90	90	88	45	0.82	12.54	*	27.35	*	*
double_v-120	120	118	60	1.85	41.65	*	13.92	*	*
double_v-150	150	148	75	3.48	20.97	*	29.54	*	*
double_v-180	180	178	90	6.20	44.28	*	54.50	*	*
triple-30	30	40	20	1.07	0.79	0.50	0.41	45.17	78.75
triple-32	32	40	20	0.64	0.74	*	0.26	55.20	4.66
triple-34	34	40	/	21.10	22.77	*	0.89	74.78	5.06
triple-60	60	79	40	158.49	315.42	114.08	4.40	*	108.01
triple-64	64	79	/	*	379.36	*	43.88	*	76.26
grid-90	68	50	34	0.04	4.51	0.03	1.53	*	21.19
grid-91	63	50	32	0.10	*	*	0.92	*	16.60
grid-92	65	50	33	0.49	*	*	0.87	*	17.40
grid-93	58	50	29	0.13	2.68	*	1.75	*	13.41
grid-94	66	50	33	0.04	3.66	*	1.61	*	*
grid-95	60	50	30	0.02	3.90	0.48	0.97	*	18.34
grid-96	62	50	31	0.07	3.30	*	0.87	*	13.62
grid-97	64	50	32	0.02	3.67	*	0.86	*	17.90
grid-98	59	50	30	0.03	*	*	1.19	*	12.30
grid-99	65	50	33	0.03	*	202.48	1.16	*	20.35

Table 4.3: PUP results for the case with *interUnitCap* = *unitCap* = 2 [ADF+11]

problem size increases on the double_v* instances. However, CLINGO performs significantly better on the grid-* instances. For the CSP encoding the variable ordering is the key to the good results and without the variable ordering the CSP model performs quite poorly. The absence of a similar variable ordering mechanism for both ASP and SAT in the experiments might explain the surprising superiority of CSP on most of the benchmarks.

DECPUP performs very good on the double-* instances, but it disappoints on other types of

Name	Zones	Sensors	Units	CSP	SAT	ASP	CBC	CPLEX
triple-30	30	40	20	0.12	2.40	0.40	182.91	24.79
triple-32	32	40	20	0.14	1.91	0.66	270.27	20.84
triple-34	34	40	20	*	1.98	0.60	331.29	*
triple-60	60	79	40	0.52	*	11.07	*	*
triple-64	64	79	40	*	*	7.61	*	*
triple-90	90	118	59	1.50	401.44	332.34	*	*
triple-120	120	157	79	3.37	*	*	*	*
grid-1	79	100	50	*	78.19	31.45	*	*
grid-2	77	100	50	*	90.89	18.91	*	*
grid-3	78	100	50	*	88.87	25.72	*	*
grid-4	80	100	50	*	95.12	24.66	*	*
grid-5	76	100	50	*	454.42	48.88	*	*
grid-6	78	100	50	*	204.85	9.15	*	*
grid-7	79	100	50	*	112.36	12.89	*	*
grid-8	78	100	50	*	*	11.89	*	*
grid-9	76	100	50	*	91.62	19.71	*	*
grid-10	80	100	50	*	545.16	13.54	*	*

Table 4.4: PUP results for the case with *interUnitCap* = 4 and *unitCap* = 2 [ADF+11]

instances. The IP encoding is not fully competitive and it particularly performs poorly on the double_v-* instances. In general, the commercial CPLEX is at least one order of magnitude faster than the open source CBC. It is also interesting to compare the double-* with the double_v-* instances, as the latter are obtained from the former by adding constraints. Both CLINGO and MINISAT succeed from the usage of the additional constraints, contrary to ECLIPSE , CBC, CPLEX and DECPUP.

The results presented in Table 4.4 show that, if CLASPFOLIO is not used to configure parameters of CLINGO, then the two DPLL-based programs, i.e. ASP and SAT, again perform quite similar, CLINGO slightly outperforming MINISAT (results are not shown). With machine learning CLINGO is clearly the winner, with the main benefits stemming from the usage of the VSIDS heuristic instead of the default BerkMin heuristic and from the exploiting local restarts.

Notably, the CSP encoding disappoints in this case, maybe because the same variable ordering is used as for the case *interUnitCap* = 2 which might be insufficient when tracking the connections between units. The IP encoding using CBC and CPLEX is not working well. Of course, any conclusions based on our experimental results have to be qualified by the following remark: every of the used KRR allows many different problem representations. Although all encodings were created in approximately the same amount of time, there is no guarantee that our problem representations are

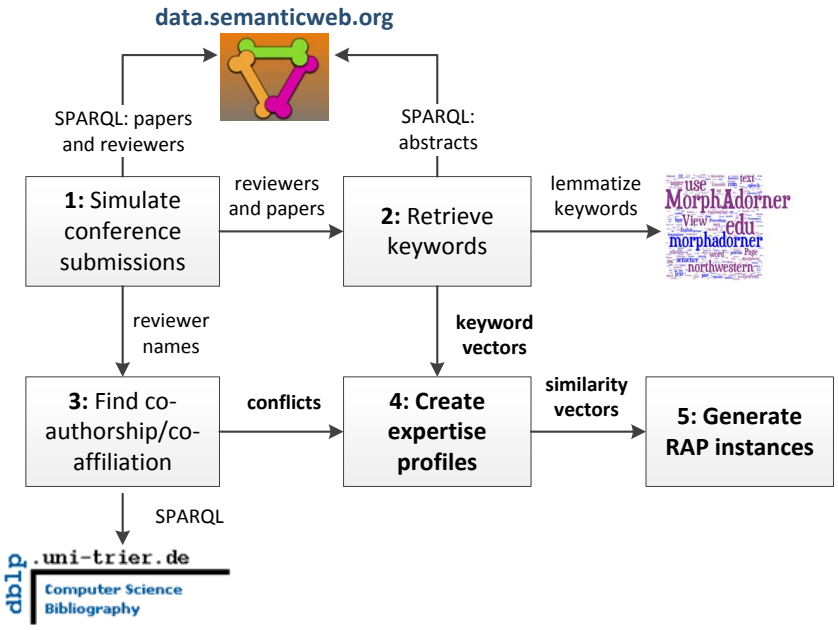

Figure 4.18: Generation of the RAP instances

the best ones possible.

Generally, the superiority of ASP solvers for the PUP with *interUnitCap* > 2, which corresponds to the general case, allows us to conclude that modern ASP solvers are worth to be considered for modeling and solving industrial configuration problems. In the next sections we are going to introduce two other practically interesting (re)configuration problems using logic-based language suggested for (re)configuration in Chapter 3 and evaluate them using Potassco.

4.3.2 Evaluating Reviewer Assignment Problem instances

To evaluate our approach for the Reviewer Assignment Problem (RAP) we generated a set of the (re)configuration instances as shown in Figure 4.18. We used Linked Data [HB11] to extract valuable information about connections between authors, such as recent co-authorship, joint affiliation or create expertise profiles. The first two types of connections allow automatic recognition of conflicts of interests. The profiles can be used to compare abstracts or keywords of published papers to submissions, thus determining the level of expertise.

Generation of instances

For a proof-of-concept implementation we have selected a fictitious set of reviewers composed of persons mentioned at `data.semanticweb.org`, as well as a subset of papers mentioned there as a fictitious set of submissions. We also retrieve information about a recent co-authorship from `http://dblp.l3s.de/d2r/` where we only link authors with unambiguous unique names present in both DBLP and `data.semanticweb.org`. In our study, we make the reasonable assumption that the more similar the paper abstract and the abstracts of a reviewer are, the more competent the reviewer is to evaluate the paper. In order to compute these similarities we extracted abstracts of submitted papers and papers written by reviewers using SPARQL queries to `data.semanticweb.org`. The set of abstracts was analyzed by established methods from information retrieval and recommender systems [JZFF10] as follows. First, we derived a list of keywords relevant to the abstracts of papers and reviewers by considering only those terms which are provided by the PC chair of a conference in form of keywords. Next, we cleaned the keywords by employing a lemmatizer such as `http://morphadorner.northwestern.edu/`. The result of this process is a term vector for each reviewer and each paper, which we use in a standard *term frequency – inverse document frequency* (TF/IDF) weighting of the paper's abstract as well as of the union of abstracts for each reviewer. The similarities of vectors describing the papers and vectors describing the reviewers are computed by the *cosine similarity measure* [JZFF10].

The similarities were used to generate the RAP instances of different size including a set of reviewers and papers as well as their bids and expertises. For each instance we applied an ASP solver to find solutions of both configuration and reconfiguration problems. The instances to the latter problem are obtained by modifying corresponding solutions of the configuration problem. The RAP modifications include situations when reviewers may drop out, papers could be withdrawn, or additional conflicts of interests may be discovered. The transformation of the legacy configuration possibly requires that some of its parts are deleted. Therefore, each reconfiguration problem instance includes requirements and transformation knowledge regarding reuse or deletion of parts of a legacy configuration.

We employ the modeling patterns described in Chapter 3 to formulate a reconfiguration problem instance[22]. The principle idea is that for every element of the legacy configuration a decision has to be made whether or not to delete or reuse this element. The reused elements are complemented

[22]SPARQL queries and ASP encoding can be found at https://sites.google.com/site/reviewersevaluation/

on demand by addition of new elements in order to fulfill all requirements. Note that in the reconfiguration case the optimization criteria of a configuration problem are extended with a criteria minimizing the costs associated with the transformation actions such as delete, reuse or create.

Evaluation

To simulate a conference bidding process we generated a set of reviewer bids in addition to the paper expertise relation presented in Section 4.1.2. For each reviewer r_i we ordered the set of papers by their similarity to reviewer r_i. As for "emulating" bidding, from the top 20% of the papers we randomly selected 3 to 7 papers and assigned to reviewer r_i a *strong willingness* to review theses papers. 4 to 8 bids of the type *weak willingness* were chosen from the next 30% of the papers and 0 to 5 conflicts were generated from the last 50%. Finally, we assign a bid of type *indifference* to all remaining papers. Authorship conflicts are taken into account during the conflict generation.

The data retrieved from the Web was used to generate configuration instances of different size in terms of numbers of papers and reviewers. Each configuration instance was solved using the approach presented in Chapter 3. Obtained solutions were then used as legacy facts in reconfiguration instances. Moreover, each instance was extended with additional facts describing reconfiguration changes such as the deletion of a paper and a reviewer as well as the declaration of a late conflict of interest. In our evaluation we differentiated between 4 possible cases: cases 1-3 are described in the previous section and the case 4 corresponds to the situation when a number of late conflicts of interest were declared and some of the reviewers already provided the reviews. The latter means that some of the made assignments cannot be changed, i.e. they have to be reused in a reconfiguration solution. Thus, for the first case we added up to 3% of assignments as late conflicts of interests. In the second case from 3 to 10% of all reviewers dropped out and papers were withdrawn. The same settings were used in the third case, which is a combination of the previous two. In the fourth case we declared that 5-10% of assignments have to be reused and provided 1-2% of late conflicts of interest. The instances were generated in such a way that the optimal reconfiguration costs for each instance was known prior to the experiment.

The evaluation results[23] presented in Table 4.5 show that the reasoner was able to find a solution for all test instances. In the instance name *TpPPPrRRR* the number *T* distinguishes between two test cases, *PPP* is the number of papers and *RRR* is the number of reviewers. In both configuration

[23]The evaluation was performed using the grounder GRINGO (v. 3.0.3) and the solver CLASP (v. 2.0.4) from POTASSCO ASP collection on a system with Intel i7-3930K CPU (3.20GHz), 32GB of RAM and running Ubuntu 11.10.

Instance	Configuration								Reconfiguration									
	avg	bounds		Optimization criteria					Reconf.	avg	bounds		Optimization criteria					
	papers	min	max	P1	P2	P3	P4	P5	case	papers	min	max	PR	P1	P2	P3	P4	P5
0p70r45	5	4	5	115	66	162	13	96	1	5	4	5	4	114	66	161	13	95
1p70r45	5	4	5	128	59	169	10	80	3	5	4	5	23	132	50	160	10	70
0p90r65	5	4	5	136	72	172	10	109	3	5	4	5	27	146	62	178	7	100
1p90r65	5	4	5	143	62	178	11	101	1	5	4	5	5	144	62	178	11	102
0p110r57	6	5	6	209	68	267	15	123	1	6	5	6	6	210	67	266	16	123
1p110r57	6	5	6	186	87	268	22	108	2	6	5	6	21	175	77	246	19	104
0p150r81	6	5	6	242	80	311	18	93	2	6	5	6	28	235	70	295	14	97
1p150r81	6	5	6	247	83	307	11	89	3	6	5	6	44	251	64	306	10	86
0p210r114	6	5	6	323	136	408	30	137	3	6	5	6	67	334	110	411	20	142
1p210r114	6	5	6	333	98	417	35	85	1	6	5	6	12	334	99	420	36	88
0p225r130	6	5	6	360	128	414	28	75	1	6	5	6	13	362	127	414	28	76
1p225r130	6	5	6	356	119	409	30	70	2	6	5	6	33	352	114	408	28	72
0p270r147	6	5	6	414	136	553	20	108	1	6	5	6	12	417	134	554	20	110
1p270r147	6	5	6	402	148	518	37	69	3	6	5	6	**82**	420	113	495	30	97
0p300r163	6	5	6	463	174	608	42	118	2	6	5	6	**90**	467	155	575	39	156
1p300r163	6	5	6	465	172	610	35	102	4	6	5	6	11	467	171	610	36	105

Table 4.5: Evaluation results for configuration and reconfiguration scenarios of the RAP

and reconfiguration cases the algorithm started the evaluation with bounds for the number of paper assignments per reviewer as described in Section 3.2. The resulting balancing bounds indicated in Table 4.5 were obtained with a timeout set to 180 seconds for checking the balancing criterion. If for given bounds the solver does not provide a solution, the bounds are relaxed. In all cases the upper bound corresponds to the average number of papers per reviewer. For the depicted balancing bounds we obtained the best configuration solutions that can be computed within a timeout period of 900 seconds; proving optimality for such (re)configuration instances seems to be infeasible in practice. The number of violations of each optimization criteria mentioned in Section 4.1.2 for each best solution is indicated in Table 4.5. The performed experiments have realistic number for PC members and submissions comparable with e.g. the last ISWC conferences from which we took the data. For the reconfiguration problem instances the solver was able to find solutions with optimal reconfiguration costs in all but the two biggest cases *1p270r147*, and *0p300r163*. A solution with optimal reconfiguration costs was usually identified by the solver in the first 10 seconds of the solving process excluding the grounding time. For the two cases mentioned above (showed in bold), the solver found solutions which reconfiguration costs are 20% and 8% higher than the optimum. The obtained results show that the proposed method is feasible for realistic Reviewer Assignment Problem instances.

Related work

The problem of assigning paper submissions to reviewers is known widely and studied by several researchers mainly from two perspectives. One branch deals with the automatic generation of the relations between reviewers and papers. The second branch investigates methods for computing the assignment between reviewers.

For the automatic generation of the expertise relation, content-based recommendation techniques were applied. Such methods are using different classification algorithms such as latent semantic indexing [DN92] or vector space models [YF99]. We leverage this approach by exploiting Linked Data and Semantic Web technology to extract data which we use to compute the expertise relation based on a vector space model. In addition, the reviewers bids might be extended by recommender system techniques [CKR09] which was out of the scope of this thesis.

The proposals for the automatic computation of the assignment relation employ different strategies depending on the constraints. The most popular problem solving algorithms are based on network flow models [HWC99, GS07] or (mixed)integer programming, e.g. [GKK+10]. These approaches assume just one weighted relation between papers and reviewers (either expertise or bids) with the exception of [GS07], whereas our work considers both relations and proposes the application of the stable marriage property. Flach et al. [FSG+09] suggest an approach to the RAP that is similar to ours, except bid initialization, i.e. our system works as an add-on of a conference management system that does not changes its behavior. However, such initialization could improve our results as well, since the opinion of a reviewer is influenced by a recommendation and becomes biased to the scores stored in the system. The score calibration method described in [FSG+09] can help PC chairs to make a final decision on a paper.

Note that, in none of the works mentioned above the reconfiguration of reviewer/paper assignments has been considered. Using an ASP framework we can model these problems succinctly. Basically, one could view reconfiguration as a form of belief revision or updates of a knowledge base (e.g. [DSTW08, SL10]), since facts about the legacy configuration must be revised. The central idea of belief revision is to apply operators to a knowledge base and to define the semantics of these operators either based on syntactical characterizations, such as defining preferred changes of the axioms of the knowledge base, or by criteria based on logical models. In reconfiguration we assume that the new knowledge base is consistent and therefore adding a legacy configuration never

leads to an inconsistent knowledge base because in the worst case the complete legacy configuration can be deleted. Consequently, a central point of belief revision, i.e. dealing with inconsistent updates, is not present in our domain. Furthermore, in belief revision the goal is to design general change operators based on some first principles (e.g. the AGM postulates) whereas in reconfiguration the knowledge engineer specifies by logical descriptions and a cost function which changes to the legacy configuration are allowed and which changes are preferred. Our experiments show that the reviewer (re)assignment task, leveraging Open Data and deploying methods for reconfiguration using SPARQL [PS08] and ASP, can efficiently be applied in practice [RPF+12].

4.3.3 Experimental study for the House problem

Finally, in this section we provide our evaluation results for the House (re)configuration problem mentioned in Section 3.1 and Section 4.1.3. At the beginning of the section we provide initial evaluation results using ASP modeling patterns suggested in Chapter 3. Then, we discuss how the House problem can be modeled using constraint programming (CP) and evaluate our constraint model using a state-of-the-art CP solver. Additionally, we investigate some ASP features such as application of a symmetry breaking tool and incorporation of domain-specific knowledge into the ASP model in order to improve the performance and the quality of solutions, and analyze the obtained results.

Preliminary evaluation

In our experiments[24] we evaluated 32 problem instances of 4 types described in Section 4.1.3. From all possible reconfiguration costs (creation, deletion and reuse) we considered only creation costs for newly generated cabinets and rooms because these are the dominant costs for our application domain. We employ the general schema based on logical descriptions for reconciliation problems suggested in [FRF+11a] and the corresponding ASP encoding is given in Appendix B.1.

In Figure 4.19 we present the performance of the solver applied to the set of reconfiguration problem instances. CLINGO was able to find optimal solutions within 600 seconds for all instances of *Newroom* and *Swap* scenarios. Optimal solutions were also found for small and mid-size instances of *Empty* scenario. For all other instances at least one suboptimal solution was found. *Long* scenario included the hardest problem instances. The solver did not find any solutions in 600 seconds for one

[24] We performed our evaluation using CLINGO (v. 3.0.3) on Core2 Duo 3Ghz with 4GB RAM.

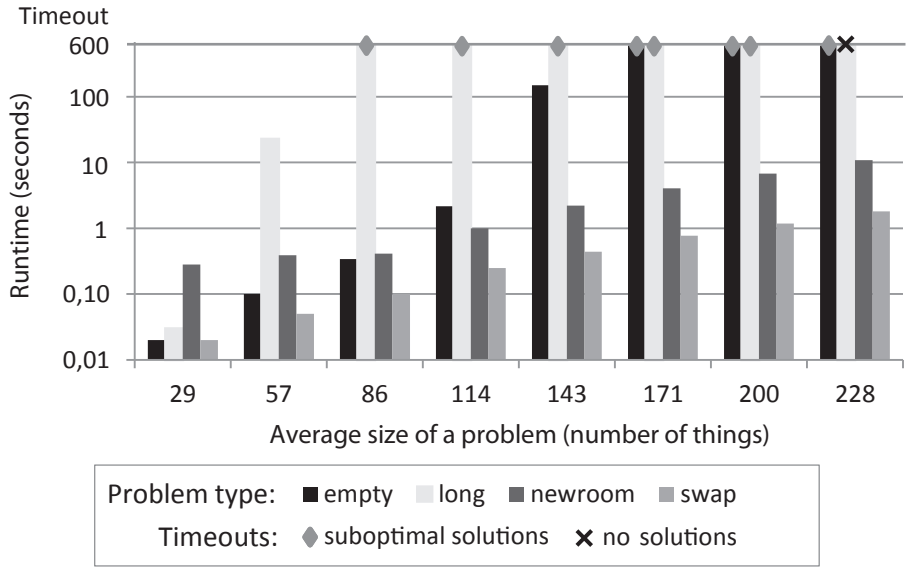

Figure 4.19: Evaluation results for the House (re)configuration problem

of them (long_2_p16t240c3). This was the only unsolved problem instance in the whole experiment.

In general, the suggested method is based on ASP which provides sound as well as complete search and optimization services. It allows to find a(n optimal) reconfiguration as well as a set of required operations (create, reuse, delete). However, some test cases were observed to be hard for the solver. Therefore, performance improvements should be investigated, e.g. generation of problem-relevant heuristics allowing to speed-up the solving process or application of symmetry breaking constraints.

Symmetry breaking

In this work we additionally researched the state-of-the-art ASP symmetry breaking approaches which can be automatically applied to our general representation language for (re)configuration problems [FRF+11a]. Namely, we tested application of SBASS [DTW11], see Section 4.2.2 for details about this tool. We performed our experiments, first, with default settings when all generators have to be computed. And we limited search of generators by a constant then. We set the constant to 5, 10 and 20 during the evaluation to see how it influences performance.

The overall evaluation results are presented in Table 4.6. “TO” indicates a timeout within 600

Instance	Optimum	No SBASS	SBASS, default	SBASS, limit=5	SBASS, limit=10	SBASS, limit=20
empty_p05t025	50	50/0:00.047	50/0:00.079	50/0:00.053	50/0:00.057	50/0:00.079
empty_p10t050	100	100/0:00.284	100/0:01.049	100/0:00.321	100/0:00.368	100/0:00.465
empty_p15t075	150	150/0:00.977	150/1:17.148	150/0:01.149	150/0:01.344	150/0:01.614
empty_p20t100	200	200/0:04.369	200/-	200/0:05.718	200/0:05.233	200/0:39.575
empty_p25t125	250	250/1:04.125	TO	250/0:58.110	200/1:09.729	250/-
empty_p30t150	300	300/-	TO	300/-	300/-	300/-
empty_p35t175	350	350/-	TO	350/-	350/-	350/-
empty_p40t200	400	400/-	TO	400/-	400/-	TO
long_2_p02t030c3	0	0/0:00.082	0/0:0.121	0/0:00.083	0/0:00.081	0/0:00.113
long_2_p04t060c3	0	0/0:00.721	0/0:01.556	0/0:00.786	0/0:01.384	0/0:01.639
long_2_p06t090c3	0	0/2:07.973	0/0:36.373	0/1:03.695	0/1:26.435	0/0:12.070
long_2_p08t120c3	0	35/-	35/-	40/-	30/-	30/-
long_2_p10t150c3	0	45/-	55/-	55/-	15/-	70/-
long_2_p12t180c3	0	90/-	75/-	80/-	85/-	80/-
long_2_p14t210c3	0	TO	150/-	TO	TO	170/-
long_2_p16t240c3	0	TO	TO	TO	TO	TO
newroom_p02t024c3	10	10/0:00.057	10/0:00.073	10/0:00.060	10/0:00.065	10/0:00.073
newroom_p04t048c3	20	20/0:00.398	20/0:00.541	20/0:00.483	20/0:00.441	20/0:00.446
newroom_p06t072c3	30	30/0:01.152	30/0:02.398	30/0:01.336	30/0:01.369	30/0:01.526
newroom_p08t096c3	40	40/0:02.793	40/0:08.380	40/0:03.350	40/0:03.485	40/0:03.794
newroom_p10t120c3	50	50/0:05.494	50/0:22.365	50/0:07.146	50/0:07.295	50/0:07.850
newroom_p12t144c3	60	60/0:10.073	60/0:53.058	60/0:13.488	60/0:13.771	60/0:14.749
newroom_p14t168c3	70	70/0:16.827	70/2:00.135	70/0:23.840	70/0:24.258	70/0:25.768
newroom_p16t192c3	80	80/0:24.816	80/3:52.280	80/0:38.284	80/0:38.753	80/0:41.115
swap_r02t035	0	0/0:00.038	0/0:00.064	0/0:00.046	0/0:00.048	0/0:00.048
swap_r04t070	0	0/0:00.124	0/0:00.250	0/0:00.152	0/0:00.155	0/0:00.153
swap_r06t105	0	0/0:00.279	0/0:00.964	0/0:00.343	0/0:00.344	0/0:00.351
swap_r08t140	0	0/0:00.594	0/0:02.855	0/0:00.724	0/0:00.734	0/0:00.743
swap_r10t175	0	0/0:01.112	0/0:06.655	0/0:01.328	0/0:01.278	0/0:01.324
swap_r12t210	0	0/0:01.853	0/0:16.417	0/0:02.171	0/0:02.151	0/0:02.224
swap_r14t245	0	0/0:02.721	0/0:28.879	0/0:03.406	0/0:03.353	0/0:03.496
swap_r16t280	0	0/0:04.407	0/0:48.896	0/0:05.115	0/0:05.299	0/0:05.196

Table 4.6: Evaluation results for the House reconfiguration problem using SBASS

seconds. 50/0:00.047 means that an optimal solution with costs equal 50 was found in 47 milliseconds, whereas 200/− reports that only a suboptimal solution with costs equal 200 was returned by a solver.

We evaluated[25] pure application of Clasp to our House model given in Appendix B.1 and an application of Clasp extended by symmetry breaking constraints (SBCs) generated by SBASS on a set of the House problem instances where we take only creation costs for individuals into account. Note that weight constraints and built-in optimization were not supported by SBASS. Therefore, the weight constraint of the encoding for the House problem provided in Section 3.6 was replaced by corresponding cardinality constraint and SBASS was modified to ignore optimization statement

[25]The evaluation was performed using Potassco ASP collection (Gringo (v. 3.0.3), Clasp (v. 2.0.5), SBASS including Saucy 1.0) on a system with Intel i7-3930K CPU (3.20GHz), 32GB of RAM and running Ubuntu 11.10

during its preprocessing step. It turned out that not all of the House problem instances can be solved in a given time frame although we limit a number of generators. Moreover, in only 2 cases (empty_p25t125, long_2_p06t090c3) runtime was improved and in 3 cases (long_2_p08t120c3, long_2_p10t150c3, long_2_p12t180c3) CLASP found better suboptimal solutions with SBASS. Together, there are only 5 cases from 32 were actually runtime or quality of a solution was better by adding of SBCs. We obtained the very similar results in [FSFR12] for the Bike configuration problem during the generation of test cases for the object-oriented configurator using ASP. The reason for this could be that generated SBCs for the House instances does not allow to break most of symmetries. In many cases, this is due to additional constraints [Sak09], e.g. the restriction disallowing to store things of different persons in the same cabinet.

Some other modern packages for detecting and breaking symmetries of CNF formulas are available. NAUTY[26] described in [ARMS03] is another approach to compute automorphism groups of graphs. However, experiments showed that it is not efficient enough for large sparse but for dense graphs [Sak09]. The instances of the House problem are sparse graphs and choice of NAUTY would be not justified. Junttila et al. [JK07] introduced BLISS[27] which is an enhancement of NAUTY and SAUCY. The authors showed experimentally that their approach outperforms the previous tools. However, those frameworks were not investigated in the current work.

Exploiting domain-specific knowledge

Our knowledge representation language is not restricted to ASP systems and the suggested rules can be expressed as a constraint satisfaction problem (CSP). The specification of the problem as a CSP allows to define variable and value orderings for the problem, and thus, enables the expression of domain-specific knowledge about the problems. Different constraint modeling languages can be used for representation of the House problem. In our experiments we applied the medium-level language MINIZINC[28]. A program implemented in this language is mapped a FLATZINC representation, which is a low-level language that can be interpreted by various CP solvers, e.g. GECODE[29]. This Finite Domain CP solver is one of the winners in the last CP solvers competitions. The problem solving stages of MINIZINC and GECODE are similar to ASP approach and are presented in Figure 4.20.

[26] http://potassco.sourceforge.net/labs.html
[27] http://www.tcs.hut.fi/Software/bliss/index.html
[28] http://www.minizinc.org/
[29] http://www.gecode.org/

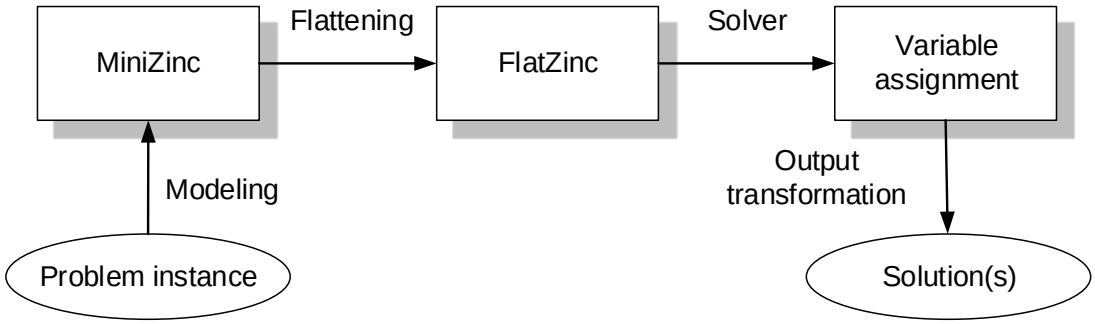

Figure 4.20: Problem solving stages of MiniZinc

The following orderings were investigated:

1. *fill cabinets first*:

   - assign things to cabinets, i.e. fix values of the variables in the array cabinetTOthing;
   - variables in the array are selected using first_fail heuristic and values indomain_min heuristic;
   - assign values to variables in roomTOcabinet, i.e. place cabinets in rooms, using the same heuristics as above;
   - the order of other variables is determined by the solver (Gecode).

2. *long things first*:

   - for each person place first longs things into cabinets and then short things;
   - after each fifth thing assignment place a cabinet into a room;
   - input_order variable selection heuristic is used to apply the ordering described above;
   - indomain_min heuristic is used to select the values;
   - the order of other variables is determined by the solver (Gecode).

We evaluated the CP encoding of the House problem on the set of (re)configuration benchmarks and compared the obtained results to those which we got using the ASP encoding, the corresponding ASP and CP encodings are provided in Appendix B.1 and Appendix C respectively. Note, that only creation costs for individuals were taken into account using both approaches and for the CP

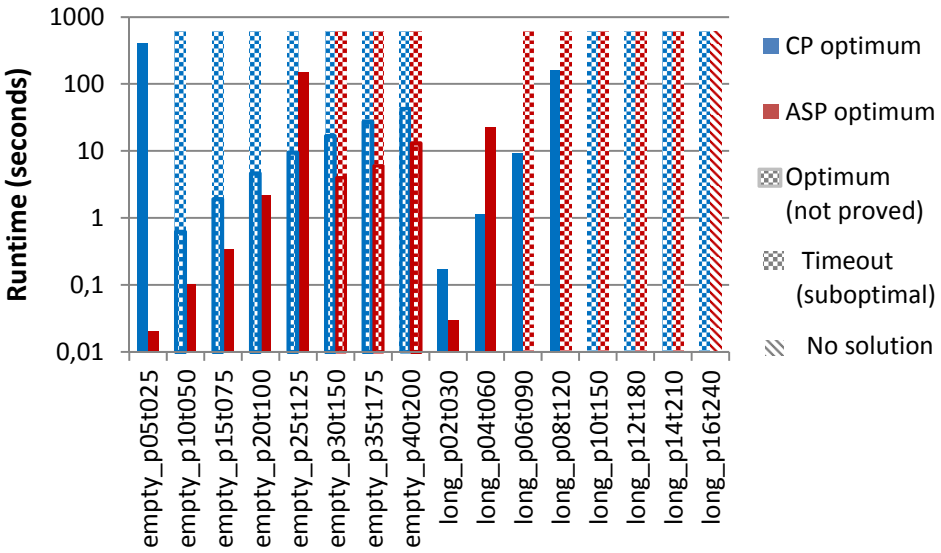

Figure 4.21: Experimental results using ASP and CP programs evaluated on Empty and Long instances

program the best working ordering was used, i.e. *long things first* for *Empty* and *Long* scenarios and *fill cabinets first* for *Newroom* and *Swap* instances.

The evaluation results for *Empty* and *Long* instances are presented in Figure 4.21 whereas the results for *Newroom* and *Swap* instances in Figure 4.22. In all experiments we used time frame equal to 10 minutes. The overall runtime using the ASP solver Clasp (v. 2.0.5) was on average better than the runtime of the CP solver Gecode (v. 3.7.3). Namely, Clasp outperformed Gecode on *Empty*, *Newroom* and *Swap* instances. However, CP allows incorporation of problem-relevant heuristics (variable and values orderings) by means of search annotations which improved the runtime for *Long* scenario. In many cases CP was able to identify the optimal model, but failed to prove the optimality whereas ASP did.

The obtained results mainly showed that modern combinatorial optimization systems are able to solve mid-sized real-world problem instances within the given time frame of 10 minutes. However, the observed performance was not satisfactory for large and complex problems of Siemens where better problem representations are required. Therefore, we analyzed domain-specific problem constrains to investigate the methods that are able to solve reconciliation problems efficiently. Since the integration of domain-specific heuristics in CP in many cases allows to speed up computations, we

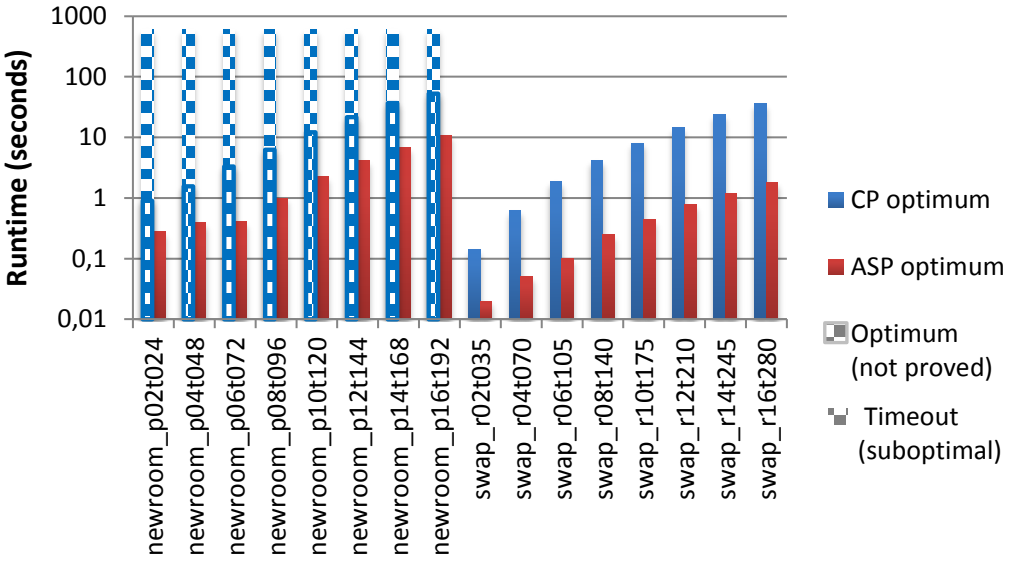

Figure 4.22: Experimental results using ASP and CP programs evaluated on Newroom and Swap instances

incorporated domain-specific knowledge described below in our ASP model.

In particular, we considered a binary constraint "per person" which disallows to place things of different persons in the same cabinets and rooms. Roughly speaking, the idea is to filter out impossible assignments of things to cabinets and rooms. The implementation of the *per person* heuristic comprises the following steps:

- take the biggest test instance of each type and deduce all facts relevant to a person with the smallest ID in a new instance (which is a part of the initial instance);
- solve this partial instance with the general program in order to obtain an optimal number of cabinets and rooms, i.e. lower bounds, required for one person;
- use the lower bounds to generate cabinets and rooms domains, called per person domains, for the initial instance;
- solve the initial instance with a modified general program where cabinet/room assignments are generated per person.

We evaluated an ASP encoding including *per person* heuristic on the House problem instances and compared the results obtained using the ASP and CP encodings described previously. The

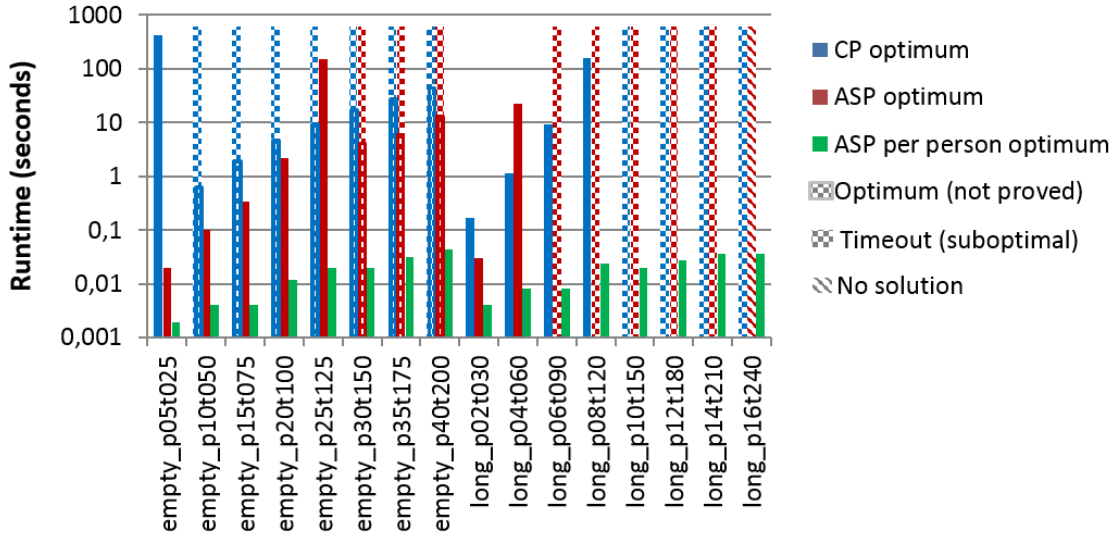

Figure 4.23: Experimental results for initial ASP and CP programs as well as for the program with "per person" heuristic evaluated on Empty and Long instances

evaluation shows that this heuristic brings a considerable improvement in both quality of a solution and solving time, see Figure 4.23 and Figure 4.24. In particular, all instances can be solved with optimal number of cabinets and rooms in less than 10 seconds each and less than 1 second on average.

The corresponding *per person* encoding of the House problem in provided in Appendix B.2. The encoding of the reconfiguration requirements, transformation rules, costs rules and optimization remains the same as in Appendix B.1 and, therefore, is omitted. In general, the integration of domain-specific heuristics in ASP is more sophisticated than in CP and it cannot be easily automated for any configuration problem. Even the integration of such heuristics in an ASP encoding requires deep knowledge of a knowledge engineer in ASP modeling.

4.4 Summary of results

The reasoning tasks for (re)configuration scenarios occurring in practice are consistency checking, completing a (partial) configuration or an optimal configuration, reconfiguration of an inconsistent configuration, finding the best reconfigurations, etc. The language based on logical descriptions suggested by Friedrich et al. [FRF+11a] and explained in Chapter 3 is compact and expressive

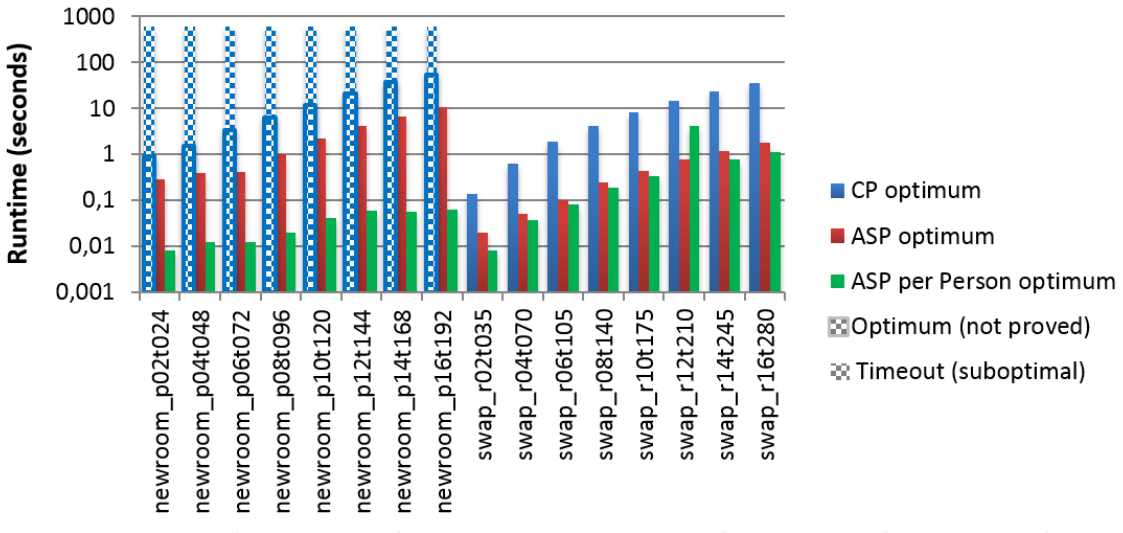

Figure 4.24: Experimental results for initial ASP and CP programs as well as for the program with "per person" heuristic evaluated on Newroom and Swap instances

enough to capture knowledge-based (re)configuration problems as it is discussed by Felfernig et al. in [HFS+14]. It can represent the configuration knowledge consisting of component types, associations, attributes, and additional constraints.

The declarative semantics of ASP programs allows a knowledge engineer to choose the order in which the rules are written in a program, i.e. the knowledge about types, attributes, etc. can easily be grouped in one place and modularized. Sound and complete solving algorithms allow to check a configuration model and support evolution tasks such as reconfiguration, see e.g. [SNTS01, FRF+11a, RPF+12]. Standard ASP frameworks support all reasoning tasks for (re)configuration except generation of individuals in domains of types and attributes, which should be provided as facts in a program. However, generation can be implemented in incremental ASP frameworks such as iClingo [GKK+08].

Our language for representing the (re)configuration knowledge and different domain costs competes with the approaches recently suggested in [FSFR12, ADV12, SFRF13]. Namely, Aschinger et al. [ADV12] propose LoCo, a general logic-based language for configuration which is a fragment of classical first-order logic. The problem specified in LoCo can easily and automatically be mapped to an ASP representation, but reconfiguration tasks are not supported in the approach. Falkner et al. [FSFR12] show how to support testing object-oriented and constraint-based configurators by

automatically generating positive and negative test cases using ASP. In [SFRF13] Schenner et al. suggest a framework, called OOASP, for the description of object-oriented product configurators using ASP which is able automatically to map object-oriented configuration knowledge bases to answer set programs and solve the different configuration and reconfiguration scenarios occurring in practice. These approaches are similar to our approach, therefore, the evaluation results on the PUP and the House problem show the same performance, when solvers with default settings are used in the experiments [FSFR12, ADV12, SFRF13].

In this chapter we have illustrated the applicability of our (re)configuration method for three practical problem cases and provided some basic modeling patterns. We evaluated the approach on a relevant set of (re)configuration problem instances provided by our industrial partner. In the experiments we aimed at finding (optimal) solutions with respect to the given costs and within a predefined time frame. ASP was chosen as the main knowledge representation formalism because of its better performance on average. We developed a variety of domain-specific heuristics for the mentioned problems, applied symmetry breaking approaches and sophisticated solving strategies (parameter tuning).

Generally, the results given above prove that ASP has limitations when applied to large scale product configuration instances. The best results in terms of runtime and solution quality were achieved in cases when domain-specific heuristics were applied. The latter have to be developed and integrated in an initial ASP encoding manually, see, for instance, Sections 4.3.1, 4.3.3. These results show that significant improvements of performance can only be achieved by application of heuristics or problem instance modifications such as introduction of symmetry breaking constraints. The main research question at this stage is how one can develop such heuristics or modifications for a problem instance automatically, i.e. without the involvement of a knowledge engineer. This will be discussed in the next chapter.

CHAPTER

5 Rewriting

A number of important real-world applications require knowledge representation (KR) languages that are able to express the existence of certain objects. For instance, a computer configuration system [Stu97] might require the existence of a compatible CPU for each motherboard. The rules with existentially quantified heads used to express these relations are often referred to as tuple generating dependencies (TGDs). Modern knowledge representation (KR) formalisms such as Datalog+/- [CGLP10] or Description Logic [BCM$^+$10] are able to represent TGDs. They are used for query answering and allow to verify whether a given set of facts is a problem solution. However, in some applications such as knowledge-based configuration the problem solutions are unknown a priori and have to be generated, e.g. by computing (subsets) of preferred logical models.

General languages for configuration problems such as LoCo [ADV12] allow to model conditional inclusion of components by means of TGDs. Since in general case a knowledge base containing TGDs might have infinite models (configurations), a language must ensure the finiteness of models by bounding the number of generated components depending on the customer input. Namely, we have to verify whether the set of customer-defined input components of a configuration problem, given as facts, suffice to make the configuration problem finite. Assuring the finiteness of models is desirable not only for guaranteeing decidability, but is also obvious for practical reasons such as realizability of a configuration since infinite configurations cannot be manufactured. After the finite bounds on the number of required components are computed, configurations are found by the means of model construction.

In many cases, computation of precise bounds is impossible since current methods [FFSS10, ADV12, Fei13] do not consider additional constraints given in a problem description. In this case one generates a number of components corresponding to an upper bound, which usually is more than

required to solve a configuration problem instance. In order to obtain a solution with the minimal number of components, a system requires a set of preference criteria to be defined by a customer. These criteria are then provided to a solver which returns preferred models. The computation of a(n optimal) model can be done by translating a general logical description of a problem instance to Answer Set Programming (ASP) [GL88, BET11].

As we showed in the previous chapter, finding an optimal solution might take unacceptable time because of the large number of constants (components) and the presence of symmetric models. They can be obtained from other models by interchanging constants substituted in existentially quantified variables. As practice shows, the larger the number of existing symmetric models is the worse is solving performance.

There are two general ways to overcome this problem: extend the knowledge base with additional symmetry breaking constraints [DTW11] and/or reduce the sets of generated constants used in the rules approximating TGDs (domain filtering [Fre91, Has93]). However, these techniques have only been developed for specific KR languages, e.g. ASP or constraint programming, and are not supported by general languages such as [ADV12]. Moreover, the evaluation presented in [Rya12] and Section 4.2.2 shows that the application of the known symmetry breaking approach for ASP [DTW11] works well for the pigeonhole problem, but does not improve solving performance when applied to such problems as rack configuration.

In this chapter we propose a novel *TGD rewriting* approach which is applicable to general knowledge representation languages. This method improves the elimination of existential quantifiers based on or-terms. The basic idea of eliminating existential quantification [ADV12] is to compute a sufficiently large set of fresh constants (called domain) for each existential quantifier and replace the existentially quantified variables in the TGDs by an or-term. Roughly speaking, the or-term contains an atom for every combination of elements of the domains of the existential quantifiers. We exploit conflicting variable substitutions in order to reduce the length of these or-terms thus reducing the number of choice points in the search space. To the best of our knowledge there are no previous proposals for an automatic rewriting of existential rules. The algorithm is evaluated on a set of industrial configuration problem instances corresponding to combinatorial problems which include several optimization criteria. These problems require selection and assignment of hardware modules depending on the customer and configuration requirements and occur during configuration of technical systems produced by Siemens.

The experimental results show that the algorithm can find (optimal) solutions for problems occurring in practice within an acceptable time. For the set of reference industrial problems, the method was able to find solutions which are up to 525% better with respect to the specified preference criteria. For the largest problem instances the standard solver was not able to find a solution with optimal costs using the original program in 3 hours, whereas the rewritten program required at most 11 minutes.

In Section 5.1 we present an approach for rewriting TGDs that allows the search to be performed efficiently. In Section 5.2 the implementation details are provided and in Section 5.3 the evaluation results are analyzed.

5.1 Rewriting of existential rules

In this section, after an introduction of some preliminaries, we describe the rewriting algorithm that uses binary constraints to limit the number of generated constants (nulls) and, thus, accelerates evaluation of programs containing TGDs.

5.1.1 Preliminaries

In configuration languages applied by Siemens in practice variants of TGDs are used to express relations between two components. Formally, a TGD σ of such a language is a first order formula of the form

$$\forall X \in S^X \; \Phi(X) \rightarrow \exists_l^u Z \in S^Z \; \Psi(X, Z) \tag{5.1}$$

where $\Phi(X)$ is an atom and $\Psi(X, Z)$ is a conjunction of atoms. An *atom* is an expression of the form $r_i(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and r_i is an element of a finite set of relation names (predicates) $\mathcal{R} = \{r_1, \ldots, r_n\}$. Each *term* can be either a variable or a constant or a null (Skolem constant). The infinite countable domains of the terms are denoted by Δ_V, Δ_C and Δ_N respectively and the union of these domains by Δ. An atom $r_i(t_1, \ldots, t_n)$ is called *ground* if all terms in the tuple $\langle t_1, \ldots, t_n \rangle$ are elements of the set $\Delta_C \cup \Delta_N$. The set $Hbase(\Delta_C \cup \Delta_N)$ (Herbrand base) contains all ground atoms that can be generated using predicates in $\mathcal{R}$ and terms in $\Delta_C \cup \Delta_N$.

Formula (5.1) includes two extensions w.r.t. classical first-order formulas, namely, counting existential quantifiers and sorts $S \subseteq \Delta_C \cup \Delta_N$. The latter is required by the fact that all components

of one type in a component catalog must have a unique identifier. Consequently, configuration languages employed in industry and proposed in [ADV12] allow the definition of a *sort* of identifiers for each component type of a problem. All sorts defined in a problem description are required to be mutually disjoint, i.e. $S_i \cap S_j = \emptyset$. As it is shown in [ADV12] a program including rules of the form (3.1) can be reduced to a classical first-order program.

An example of a typical configuration TGD is a binary relation between two components C_1 and C_2 that relates at least l and at most u instances of C_2 with each C_1:

$$\forall X \in S^X \; C_1(X) \rightarrow \exists_l^u Z \in S^Z \; C_2(Z) \wedge C_1 2 C_2(X, Z)$$

where sorts S^X and S^Z for variables X and Z contain identifiers of component instance available in a problem description.

An atom $\Phi(X)$ is often called the *body* of a rule and the conjunction $\Psi(X, Z)$ is the *head*. A rule with an empty body $body(r) = \emptyset$ is usually referred to as a *fact*. Facts can be of the two types *existential* and *ground* depending on whether they contain some existentially quantified variables or not. Note that, an existential fact can be a conjunction of atoms including one existentially quantified variable. A program Σ is a finite set of rules. It contains rules of the form (5.1) as well as first-order rules with only universally quantified variables. In addition, $edb(\Sigma)$ denotes a set of all ground facts of the program Σ and by $idb(\Sigma)$ all other rules.

A *substitution* is a homomorphism $\theta : \Delta \mapsto \Delta_C \cup \Delta_N$ that maps elements of Δ_C and Δ_N to themselves. In order to simplify the presentation we denote a substitution of the variables in an atom at by $\theta(at)$. For the sorted variables the substitution function is defined as $\theta : \Delta \mapsto S$, where $S \subseteq \Delta_C \cup \Delta_N$.

Since sorts and counting existential quantifies can be reduced to classical first-order logic we use the following semantic of a program Σ. Let $\mathcal{A}$ be a set of atoms. Given an atom at the set $\mathcal{A}$ entails at ($\mathcal{A} \models at$) if there is a substitution θ such that $\theta(at) \in \mathcal{A}$. A set of ground atoms $\mathcal{M} \subset Hbase(\Delta_C \cup \Delta_N)$ is a *model* of the program Σ if for every rule $\sigma \in \Sigma$ there exists a substitution $\theta(body(\sigma)) \subseteq \mathcal{M}$ such that $\mathcal{M} \models \theta'(head(\sigma))$, where θ' contains all and only substitutions for existentially quantified variables Z, see [FKP05, CGLP10, LMTV12] for more details. In configuration solutions often correspond to a minimal set of atoms in order to reduce the costs. Therefore, in practice we can focus on the computation of minimal models. The computation of finite models containing the optimal solutions (configurations) can be accomplished by using ASP encodings and solvers.

Note, that in case of configuration the number of nulls used in a solution must be finite, since infinite solutions are of no practical interest. Therefore, configuration languages use methods as in [Fei13] to compute the required number of nulls for each sort or to determine that there is no finite solution. Consequently, we assume that the sorts are fixed. The bounds can be determined for the number of components by methods presented in [FFSS10, ADV12, Fei13]. For the practical configuration problems of Siemens these bounds can be computed in polynomial time.

5.1.2 Conflict-based program rewriting

Given a program Σ, a reasoning algorithm, e.g. chase [FKP05, CGLP10], usually starts from the set of rules $edb(\Sigma)$ and iteratively extends it by searching any applying substitutions to the rules in $idb(\Sigma)$. In case of a TGD σ with $\theta(body(\sigma)) \subseteq edb(\Sigma)$ the reasoning algorithm first rewrites it as an existential fact of the form

$$\exists_l^u Z \in S^Z \; \Psi(\theta(X), Z) \tag{5.2}$$

where $\theta(X)$ maps variable X to some constant in S^X. Next, the algorithm searches for an *extension* θ' of the substitution θ on $X \cup Z$ associating the variable Z with some element of S^Z. The resulting ground facts are then added to $edb(\Sigma)$. Usually reasoning algorithms use a variant of an extension function θ' that associates a *fresh* null with each variable in order to obtain a universal solution $\mathcal{M}$, i.e. such solution that any other solutions, say $\mathcal{M}'$, can be obtained from $\mathcal{M}$ by a homomorphism $h : \Delta_N \cup \Delta_C \rightarrow \Delta_N \cup \Delta_C$ that maps elements of Δ_C to themselves [FKP05, LMTV12].

Example 3. Consider the following program capturing a frequent case in technical configuration which includes two component types and a typical $\exists_1^1$ relation between them. Given a set of things, store each of them in exactly one cabinet taking into account that things t_1 and t_2 cannot be placed in the same cabinet. The domain of cabinets is defined as $S^Z = \{\varphi_1, \varphi_2, \varphi_3\}$.

$$\begin{aligned} r_1 : &\quad thing(t_1) \wedge thing(t_2) \wedge thing(t_3) \\ r_2 : &\quad \forall X \; thing(X) \rightarrow \exists_1^1 Z \in S^Z \; t2c(X, Z) \\ r_3 : &\quad \forall X \; t2c(t_1, X) \wedge t2c(t_2, X) \rightarrow \end{aligned}$$

The reasoning algorithm starts with an $edb(\Sigma) = \{thing(t_1), thing(t_2), thing(t_3)\}$ and finds a

substitution $\theta_1 = \{X/t_1\}$. This substitution allows to rewrite the second rule as an existential fact $\exists^1_1 Z \in S^Z\ t2c(t_1, Z)$. The second substitution $\theta_2 = \{X/t_2\}$ results in the fact $\exists^1_1 Z \in S^Z\ t2c(t_2, Z)$ and the third $\theta_3 = \{X/t_3\}$ in $\exists^1_1 Z \in S^Z\ t2c(t_3, Z)$. An extension function might introduce different mappings. For instance, it can map every variable to a different null and obtain three ground atoms $t2c(t_1, \varphi_1)$, $t2c(t_2, \varphi_2)$, and $t2c(t_3, \varphi_3)$, where each φ_i corresponds to a cabinet. $edb(\Sigma)$ extended with these facts is a model of the program above. However, an extension resulting in $t2c(t_1, \varphi_1)$, $t2c(t_2, \varphi_1)$, and $t2c(t_3, \varphi_1)$ does not allow to obtain a model of the program because of the constraint (rule r_3).

Definition 16 (Conflict set)**.** Let $\mathbf{\Psi} := \{\Psi(\theta_1(X), Z), \ldots, \Psi(\theta_n(X), Z)\}$ be the set of sorted existential facts Ψ of a consistent program Σ in which variable Z range over the same sort S^Z. A pair of facts $CS \subseteq \mathbf{\Psi}$ is a *conflict set* iff $\Sigma \cup \{\exists Z\ \bigwedge_{a \in CS} a\}$ is inconsistent.

A set **CS** of conflict sets can be computed using this definition as shown in Section 5.2. Given the set **CS**, we can rewrite TGDs in Σ in a way that the reasoning algorithm will extend substitutions for any pair of conflicting atoms with different constants (nulls).

Definition 17 (TGD rewriting problem)**.** Let σ be a TGD of the form (5.1) in a consistent program Σ that can be rewritten as a set of existential facts $\mathbf{\Psi}$ using $edb(\Sigma)$. In addition, let **CS** be a set of conflict sets w.r.t. the set $\mathbf{\Psi}$. The TGD rewriting problem is to find a set of domain restrictions $SN = \left\{\left\langle D^X_1, D^Z_1\right\rangle, \ldots, \left\langle D^X_n, D^Z_n\right\rangle\right\}$ for the substitution function θ, where $D^X_i \subseteq S^X$ and $D^Z_i \subseteq S^Z$ for $1 \leq i \leq n$, such that the original TGD $\sigma \in \Sigma$ can be rewritten as a set of TGDs

$$\overline{\sigma} = \left\{\forall X \in D^X_i\ \Phi(X) \rightarrow \exists^u_l Z \in D^Z_i\ \Psi(X, Z) \mid \langle D^X_i, D^Z_i\rangle \in SN\right\} \tag{5.3}$$

and the following conditions hold:

a) the set $\mathbf{\Psi}$ of sorted existential facts of the program $edb(\Sigma) \cup \overline{\sigma}$ does not contain conflict sets $CS \in$ **CS**;

b) the program $edb(\Sigma) \cup \overline{\sigma}$ preserves all possible models of the program $edb(\Sigma) \cup \{\sigma\}$ up to symmetric ones that can be obtained by renaming of nulls.

Note that, if multiple sets of existential facts $\mathbf{\Psi} = \{\mathbf{\Psi}_1, \ldots, \mathbf{\Psi}_i\}$ ranging over mutually disjoint sorts $S^Z_1, \ldots, S^Z_i$ were derived by a reasoning algorithm for TGDs $\{\sigma_1, \ldots, \sigma_i\} \subseteq \Sigma$, then the TGD rewriting problem is solved for each $\mathbf{\Psi}_k \in \mathbf{\Psi}$ separately.

Algorithm 1: GenerateExtensions

input : A consistent program Σ, a set $\mathbf{\Psi}$ of existential facts, a set **CS** of conflict sets, a domain S

output: A set of substitutions associated with a set of nulls SN

1 $SN \leftarrow \emptyset$;
2 **for** $i \leftarrow 1$ **to** $|\mathbf{\Psi}|$ **do**
3 $\quad D^X \leftarrow \textsc{getSubstitutionConstants}(\Psi_i)$;
4 $\quad D^Z \leftarrow \textsc{getConstants}(\Sigma, S)$;
5 $\quad D^Z \leftarrow D^Z \cup \textsc{getNulls}(\Psi_i, S)$;
6 $\quad$ **for** $j \leftarrow 1$ **to** $i - 1$ **do**
7 $\quad\quad$ **if** $\forall CS \in \mathbf{CS} \quad CS \notin \left\{\Psi_i, \Psi_j\right\}$ **then**
8 $\quad\quad\quad D^Z \leftarrow D^Z \cup \textsc{getNulls}(\Psi_j, S)$;
9 $\quad SN \leftarrow SN \cup \left\{\left\langle D^X, D^Z\right\rangle\right\}$;
10 **return** SN;

To solve the TGD rewriting problem, we suggest Algorithm 1 which takes the following input: (a) a consistent program Σ; (b) a set of existential facts $\mathbf{\Psi} := \{\Psi(\theta_1(X), Z), \ldots, \Psi(\theta_n(X), Z)\}$ derived by a reasoning algorithm for a TGD $\sigma \in \Sigma$; (c) a set of conflict sets **CS** for $\mathbf{\Psi}$ and (d) a finite set of fresh nulls S that comprises enough elements to obtain a solution. Given a correct input, the algorithm finds a tuple $\left\langle D_i^X, D_i^Z\right\rangle$ for each existential fact $\Psi_i \in \mathbf{\Psi}$, where the set $D_i^X \subseteq S^X$ contains all constants that are substituted by θ_i to the variable X in Formula (5.1) and the set $D_i^Z \subseteq S^Z$ contains all nulls which are used to substitute Z. Next, given the resulting set of all tuples $SN = \left\{\left\langle D_1^X, D_1^Z\right\rangle, \ldots, \left\langle D_n^X, D_n^Z\right\rangle\right\}$ we generate TGDs of the form:

$$\forall X \in D_i^X \; \Phi(X) \rightarrow \exists_l^u Z \in D_i^Z \; \Psi(X, Z) \tag{5.4}$$

for each tuple $\left\langle D_i^X, D_i^Z\right\rangle \in SN$ and add them to the program Σ instead of the original TGD.

Our algorithm uses three functions: getSubstitutionConstants, getNulls, and getConstants. The first function retrieves all constants that were substituted for the universally quantified variable X. getConstants returns all constants from the set $S \cap \Delta_C$ contained in Σ. Finally, the injective function $\textsc{getNulls} : \mathbf{\Psi} \mapsto \mathcal{P}(S \cap \Delta_N)$ associates a set of nulls $\boldsymbol{\varphi}_i \subseteq S \cap \Delta_N$ with every $\Psi_i \in \mathbf{\Psi}$, such that $\boldsymbol{\varphi_i}$ includes a fresh null for the variable Z, which is not used in any of the previous substitutions. Note that Algorithm 1 prevents the generation of many symmetric solutions by generating only a lower triangle of the matrix of nulls that can be substituted for existentially quantified variables (line 6).

Each row of this matrix corresponds to an element of D^X and each column to one of the nulls in D^Z. For instance, if atoms $t2c(t_1, Z)$ and $t2c(t_2, Z)$ are not conflicting then the algorithm generates two tuples $\langle \{t_1\}, \{\varphi_1\} \rangle$ and $\langle \{t_2\}, \{\varphi_1, \varphi_2\} \rangle$, thus, avoiding a symmetric solution $\{t2c(t_1, \varphi_2), t2c(t_2, \varphi_2)\}$.

Theorem 1. *Algorithm 1 always terminates in at most $O(|\mathbf{\Psi}|^2)$ steps.*

Proof. The algorithm always terminates since its input is defined by a finite program Σ as well as finite sets **CS**, $\mathbf{\Psi}$ and S. Therefore, the functions getSubstitutionConstants, getNulls, and getConstants always return finite sets of constants for any possible input. The number of steps made by the algorithm depends solely on the cardinality of the set $\mathbf{\Psi}$ (lines 2 and 6). Let $n = |\mathbf{\Psi}|$, then the total number of steps is determined as:

$$0 + 1 + 2 + \cdots + (n-1) = \sum_{i=1}^{n} (i-1) = \frac{n(n-1)}{2} \in O(n^2)$$

Consequently, the algorithm has an order of $|\mathbf{\Psi}|^2$ time complexity. □

Now we prove that Algorithm 1 allows to find a set *SN* such that the rewritten TGDs satisfy the two conditions given in Definition 17.

Theorem 2. *Algorithm 1 returns a set SN such that the set $\mathbf{\Psi}$ of sorted existential facts of the program $edb(\Sigma) \cup \overline{\sigma}$ does not contain conflict sets $CS \in$ **CS**.*

Proof. For each atom $\Psi_i \in \mathbf{\Psi}$ Algorithm 1 initializes the set D^Z with a set of fresh nulls $\vec{\varphi}_i$ corresponding to the fact Ψ_i (line 4) and all constants of S appearing in Σ (line 3). These constants cannot result in generation of any conflict set (Definition 16), since the program Σ is consistent. Next (lines 7 and 8), D^Z is extended with the nulls corresponding to all existential facts $\Psi_j \in \mathbf{\Psi}$ that are non-conflicting with Ψ_i, i.e. $\forall CS \in$ **CS** $CS \not\subseteq \{\Psi_i, \Psi_j\}$. Therefore, none of the rewritten TGDs $\sigma_i, \sigma_k \in \overline{\sigma}$ corresponding to the conflicting existential facts Ψ_i, Ψ_k do not share any nulls, i.e. $D_i^Z \cap D_k^Z \cap S = \emptyset$. The rewriting approach modifies only the range of the substitution function, defined as a homomorphism $\theta : \Delta \mapsto S$ by replacing S with its subset D^Z. All other elements of the structure such as predicates remain constant. Consequently, no pair of existential facts $\Psi_i, \Psi_j \in \mathbf{\Psi}$ of the program $edb(\Sigma) \cup \overline{\sigma}$ comprises any element of the set of conflict sets **CS**. □

Theorem 3. *The set of extensions SN generated by Algorithm 1 results in a sound and complete rewriting of the original TGD σ as $\overline{\sigma}$ according to Definition 17:*

- *Rewriting is* sound *if all models of the rewritten program* $edb(\Sigma) \cup \overline{\sigma}$ *are also models of the program* $edb(\Sigma) \cup \{\sigma\}$.
- *Rewriting is* complete *if the set of models of the rewritten program* $edb(\Sigma) \cup \overline{\sigma}$ *coincides with the set of models of the program* $edb(\Sigma) \cup \{\sigma\}$ *up to symmetric ones, i.e. resulting in renaming of nulls.*

Proof. (*Soundness*) For every existential fact Ψ_i Algorithm 1 constructs the sort D^Z such that $D^Z \subseteq S$. Therefore, the sort D^Z of any rewritten TGD $\sigma' \in \overline{\sigma}$ of the form (5.4) is always a subset of the sort S of the original TGD $\sigma \in \Sigma$ of the form (5.1). Consequently, every model of the program $edb(\Sigma) \cup \overline{\sigma}$ is also a model of $edb(\Sigma) \cup \{\sigma\}$.
(*Completeness*) The completeness of Algorithm 1 follows from the fact that each sort D_i^Z, constructed for an existential fact Ψ_i, comprises: (a) all constants of the input program Σ which also appear in the original sort S, (b) u fresh nulls from the set S, and (c) all nulls appearing in a sort $D_j^Z \subseteq S$ constructed for a non-conflicting existential fact Ψ_j, where $1 \leq j \leq i-1$. The exclusion of conflicting nulls results only in elimination of interpretations which are not models of Σ and does not affect the models according to Definition 16. The exclusion of fresh nulls in Algorithm 1 can only result in exclusion of those models of $edb(\Sigma) \cup \{\sigma\}$ that can be obtained by renaming of nulls substituted by θ to the existentially quantified variable Z. In case $\mathbf{CS} = \emptyset$, all sorts $D_1^Z, \ldots, D_n^Z$ constructed by Algorithm 1 are equal up to fresh nulls and the sort $D_n^Z = S$. Since the rewriting approach modifies only the range of the substitution function and does not modifies any other elements of the structure, it follows that the models of the program $edb(\Sigma) \cup \{\sigma\}$ and the program $edb(\Sigma) \cup \overline{\sigma}$ coincide up to symmetric ones. □

Of course, the method does not ensure the elimination of all possible conflicts, but only those given in **CS**. In addition, we also do not require **CS** to contain all conflicts. In particular, in our implementation we focus on conflicts which can be efficiently detected by using the Horn fragment of Σ, where consistency can be checked in polynomial time. Similar to filtering methods used in constraint satisfaction problem solving, we focus on the elimination of the binary conflicts in order to increase efficiency of the algorithm. Our evaluation shows that the suggested algorithm allows to achieve significant reduction of computation time for real-world problems.

Example 4 (continue Example 3)**.** Let $edb(\Sigma)$ be extended with facts defining two more things $\{thing(t_4), thing(t_5)\}$ and $S^Z = \{\varphi_1, \ldots, \varphi_5\}$. Also in $idb(\Sigma)$ we replace the constraint r_3 with a set

of constraints that do not allow to assign any of the things $\{t_1, t_2\}$ and $\{t_3, t_4, t_5\}$ to the same cabinet. Assume that for the set of facts

$$\mathbf{\Psi} = \{t2c(t_1, Z), t2c(t_2, Z), t2c(t_3, Z), t2c(t_4, Z), t2c(t_5, Z)\}$$

all constraints in $idb(\Sigma)$ result in the following set of conflict sets:

$$\begin{aligned}\mathbf{CS} = \{ &\{t2c(t_1, Z), t2c(t_3, Z)\}, \{t2c(t_1, Z), t2c(t_4, Z)\}, \{t2c(t_1, Z), t2c(t_5, Z)\}, \\ &\{t2c(t_2, Z), t2c(t_3, Z)\}, \{t2c(t_2, Z), t2c(t_4, Z)\}, \{t2c(t_2, Z), t2c(t_5, Z)\}\}\end{aligned}$$

Application of Algorithm 1 to the given sets $\mathbf{\Psi}$, **CS** and the domain of nulls $S^Z = \{\varphi_1, \ldots, \varphi_5\}$ results in the following set of extensions $\mathcal{SN}$:

$$t_1 : \{\varphi_1\} \quad t_2 : \{\varphi_1, \varphi_2\} \quad t_3 : \{\varphi_3\} \quad t_4 : \{\varphi_3, \varphi_4\} \quad t_5 : \{\varphi_3, \varphi_4, \varphi_5\}$$

Given the set SN we use the suggested transformation to rewrite the TGD in Example 4 as follows:

$$\begin{aligned} &thing(t_1) \rightarrow \exists_1^1 Z \in \{\varphi_1\}\, t2c(t_1, Z) && thing(t_3) \rightarrow \exists_1^1 Z \in \{\varphi_3\}\, t2c(t_3, Z) \\ &thing(t_2) \rightarrow \exists_1^1 Z \in \{\varphi_1, \varphi_2\}\, t2c(t_2, Z) && thing(t_4) \rightarrow \exists_1^1 Z \in \{\varphi_3, \varphi_4\}\, t2c(t_4, Z) \\ & && thing(t_5) \rightarrow \exists_1^1 Z \in \{\varphi_3, \varphi_4, \varphi_5\}\, t2c(t_5, Z)\end{aligned}$$

The resulting program reduces the number of possible extensions significantly. In the Example 3 the standard elimination of existential quantification would result in a set of five nulls $\{\varphi_1, \ldots, \varphi_5\}$, where each existential quantified variable can be substituted by an element of this set. Such a set of nulls allows $5^5 = 3125$ possible combinations of substitutions of nulls, whereas the rewritten program allows only 12 combinations which is 260 times smaller than using the common algorithm. In this example we preserve all possible models, up to symmetric ones, because the algorithm eliminated only those nulls from the sets D_i^Z whose substitution results in an inconsistency.

5.1.3 Rewriting of multiple TGDs

Descriptions of complex problem instances might include multiple TGDs such that some of them depend on the others, i.e. the body $\Phi(X)$ of a TGD σ_1 contains an atom $at \in \Phi(X)$ which can be

Algorithm 2: Solve

input : A consistent program Σ
output: A set of atoms representing the best solution of a problem instance

Algorithm SOLVE(Σ)
1 **return** FINDSOLUTION($\Sigma, \emptyset, \emptyset$);

Function FINDSOLUTION(Σ, Sol, TS)
1 **loop**
2 $F \leftarrow$ GETNEXTSOLUTION(Σ);
3 **if** $F = \emptyset$ **then return** Sol;
4 $TS' \leftarrow TS \cup F$;
5 $EF \leftarrow$ GETEXISTENTIALFACTS($\Sigma \cup F$);
6 **if** $EF = \emptyset$ **then** $Sol \leftarrow$ GETBESTSOLUTION(Sol, TS') ;
7 **else**
8 $SN \leftarrow \emptyset$;
9 **foreach** $\langle \mathbf{\Psi}_i, S_i \rangle \in EF$ **do**
10 **CS** $\leftarrow$ GETCONFLICTSETS($\mathbf{\Psi}_i, \Sigma \cup F$);
11 $SN \leftarrow SN \cup$ GENERATEEXTENSIONS(**CS**, $\mathbf{\Psi}_i, S_i, \Sigma$);
12 $\Sigma' \leftarrow$ REWRITE($SN, \Sigma \cup F$);
13 $Sol \leftarrow$ FINDSOLUTION(Σ', Sol, TS');

derived from some atom in the head $\Psi(X, Z)$ of a TGD σ_2.

Algorithm 2 starts with call to a recursive function FINDSOLUTION which takes tree arguments: a program Σ, a set *Sol* comprising the best solution found so far and a set *TS* comprising elements of a partial solution. The output of FINDSOLUTION is the best solution that was found for the input program Σ by the rewriting algorithm. Each call to FINDSOLUTION starts with computation of a partial solution F using GETNEXTSOLUTION function (line 3). This function returns a set of ground facts computed for a (rewritten) program excluding all TGDs of form (5.1) which bodies are not justified by facts in Σ. In the first iteration the function returns all facts that are derivable from $edb(\Sigma)$ without TGDs including existential quantification. In the next iteration, it returns a set of ground facts $\left\{\Psi(\theta'_1(X), \theta'_1(Z)), \ldots, \Psi(\theta'_n(X), \theta'_n(Z))\right\}$ computed for TGDs justified by the facts derived in the previous iteration, etc. If the next partial solution F is empty (line 3), i.e. the function enumerated all possible partial solutions, then the algorithm exits the loop and returns the best solution stored in *Sol*.

Given a non-empty set of facts F the algorithm constructs a new partial solution TS' (line 4) and

calls getExistentialFacts function (line 5). The latter retrieves a set EF of tuples $\langle \mathbf{\Psi}_i, S_i \rangle$ from Σ, where all $\mathbf{\Psi}_i$ are justified by the facts in F. In addition, the function selects only those existential facts Ψ_i that are generated by new substitutions of the universal variables. By "new" we mean those which were not used for rewriting in previous iterations. In case all TGDs are rewritten, i.e. $EF = \emptyset$ (line 6), the set TS' represents a complete solution of the problem. The algorithm compares the actual solution TS' and the best known solution of the problem Sol using getBestSolution. This function returns the best out of two solutions w.r.t. the customer preferences criteria. Note that if one of the solutions is empty, e.g. $Sol = \emptyset$, then getBestSolution always prefers the non-empty solution. If the set of existential facts is not empty, then for each tuple $\langle \mathbf{\Psi}_i, S_i \rangle$, comprising a set of existential facts $\mathbf{\Psi}_i$ and a sort S_i of existentially quantified variables occurring in $\mathbf{\Psi}_i$, we compute a set of conflicts according to the Definition 16 (line 9). Given the set of conflicts **CS** Algorithm 1 generates the set of extensions SN (line 10) and the selected TGDs are then rewritten to obtain Σ', as described above (line 11).

Next, the function findSolution calls itself providing the extended program, the best solution Sol and the current partial solution TS' as an input (line 12). The facts F, added on the rewriting step (rewrite), justify the bodies of TDGs that were not justified on the previous iteration, thus allowing us to apply the rewriting one more time. The program continues the search in a depth-first order until all solutions are enumerated and returns the best one in case preference criteria are defined by a customer.

When the rewriting method is applied to recursive TGDs we guarantee its termination because the lower and upper bounds on the number of constants corresponding to each component are finite and determined prior to execution of the Algorithm 2. To rewrite a recursive TDG we have to take into account all constants of a sort that are already appearing in a partial solution. These constants must be added to the initial set D^Z by the function getConstants (see Algorithm 1).

Theorem 4. *Let getNextSolution terminate for the input program* Σ. *Then, Algorithm 2* terminates.

Proof. The theorem can be proved by mentioning the fact that Algorithm 2 only extends a reasoning method implemented in getNextSolution with the rewriting (lines 4-11 in findSolution). The rewriting does not modify the structure, but reduces domains of the substitution function. Therefore, it cannot result in the modifications leading to non-termination of the reasoning algorithm. Moreover, all functions used in the rewriting procedure, including generateExtensions (Theorem 1), terminate. Consequently, Algorithm 2 also terminates. □

Theorem 5. *Let* getNextSolution *implement a sound and complete reasoning algorithm for the input program* Σ*. Then, Algorithm 2 is:*

- sound*, if it always returns a set of atoms Sol which is a model of* Σ*.*
- complete*, if it enumerates all models of* Σ *up to symmetric ones, i.e. always returns a preferred model of* Σ*.*

Proof. (*Soundness*) The proof follows from the fact that both reasoning algorithm (getNextSolution) and the generation of extensions (generateExtensions) are sound (Theorem 3).
(*Completeness*) Since getNextSolution is complete, it iterates over all possible partial solutions of Σ on every step of recursion. Due to the fact that our rewriting approach is complete, we conclude that the algorithm enumerates all models of Σ up to symmetric ones resulting in renaming of nulls (according to Theorem 3). Consequently, the algorithm enumerates all solutions and will find at least one preferred model of Σ. □

The algorithm allows the identification of a solution by executing its fast variant in which we investigate only the left-most branch in the search tree, i.e. it selects one of the optimal partial solutions on each step and continues until the whole model is generated. This greedy approach is important in practice since customers often want to test the model they are developing and require a solving method which responds quickly.

5.2 Implementation

The method suggested in this chapter can be applied to knowledge bases defined using knowledge representation languages such as LoCo [ADV12], thus allowing application of modern solvers of combinatorial search problems like SAT, answer set programming or constraints. We ensure the finiteness of the logical models by bounding the number of generated nulls depending on the customer input. The rewriting approach is implemented using the translation to ASP as suggested in [FRF+11a] and exemplified in Chapter 3. Following this methodology each TGD is translated to a choice rule [SNS02] of the form:

$$lower\,\{\Psi(X,Z) : domain(Z)\}\,upper \text{ :- } \Phi(X). \tag{5.5}$$

where $domain(Z)$ is a domain for the existentially quantified variables, which contains as many constants as the corresponding set D^Z. An operator ":" generates a set of atoms by substituting all possible constants appearing in the atoms with the *domain* predicate. The ASP semantics [GKS12] ensures that at least *lower* and at most *upper* atoms from the set of atoms defined in the head of a choice rule are elements of each answer set. The latter is a set of atoms justified by a program with respect to the ASP semantics [BET11]. The implementation is done using Potassco system [GKK+11b] which is the winner of the last ASP competition.

Example 5. The sample program Σ presented in Example 3 is encoded as follows:

$$thing(t1).\quad thing(t2).\quad thing(t3).\quad domain(c1).\quad domain(c2).\quad domain(c3).$$
$$1\,\{t2c(X,Z) : domain(Z)\}\,1 \;:\!-\; thing(X).$$
$$:\!-\; t2c(t1,X), t2c(t2,X).$$

Application of the Potassco grounder Gringo expands the choice rule and the constraints into three pairs of rules such as:

$$1\,\{t2c(t1,c1), t2c(t1,c2), t2c(t1,c3)\}\,1 \;:\!-\; thing(t1).$$
$$:\!-\; t2c(t1,c1), t2c(t2,c1).$$

A sample answer set justified by the ground program includes all facts given in $edb(\Sigma)$ and a set of atoms $\{t2c(t1,c1), t2c(t2,c2), t2c(t3,c1)\}$. A solver identifies 18 possible answer sets for this problem.

Minimization of the number of nulls, i.e. constants defined in the domain, used in the solution is very important. In problems like configuration this corresponds to the minimization of the number of components used in the system, thus reducing the costs of a product. In our case, minimization can be done using the optimization algorithms available in ASP systems. We can add the following rules to model this in Potassco:

$$usedNull(Z) \;:\!-\; t2c(X,Z).$$
$$\#minimize\,\{usedNull(Z)\}.$$

The latter rule specifies a preference criterion forcing a solver to return only answer sets including

a minimal number of atoms from the set. In our example, the preferred answer set is the one which includes a minimal number of elements from the set $\{usedNull(c1), usedNull(c2), usedNull(c3)\}$. Given such preference criterion a solver returns only 12 answer sets.

The rewriting program executes the algorithms presented in Section 5.1. The identification of conflict sets is implemented using Definition 16. We remove from the ground program all choice rules and add pairs of facts which represent a potential conflict set. If the resulting program is unsatisfiable then the pair is a conflict set. Note, that the translated ASP program without choice rules includes only Horn clauses (clauses with at most one positive literal). Completeness of the set of conflicts **CS** is not required. Taking into account that there are at most $n(n-1)/2$ possible pairs of atoms, the identification of conflicts can be done in polynomial time, i.e. $O(n^2)$.

In addition, performance of the solver can be improved by introduction of the ordering on the sets of fresh nulls. For the original program we specify a set of rules defining lexicographical ordering of all constants in a domain. Applied to Example 3 the ordering rules force a solver to use cabinets corresponding to lexicographically smaller constants first. We define ordering rules for nulls which are used in the rewritten program associated with each set of non-conflicting atoms, i.e. such atoms of any facts Ψ_i and Ψ_j for which $D_i^Z \cap D_j^Z \neq \emptyset$. The order is defined by the number of occurrences of a null in tuples of the set SN. The more frequently the null is used the higher is its order. In Example 4 for set $\{t_1, t_2\}$ the set of nulls is ordered as $\{\varphi_1, \varphi_2\}$ since φ_1 is used twice and φ_2 only once. Such ordering requires a solver to substitute φ_1 first.

5.3 Evaluation

We evaluated our approach on a slightly modified version of the House configuration problem proposed in [MBSF09]. The problem is an abstraction of various configuration problems occurring in practice of Siemens, where entities may be contained in other entities and several requirements and constraints in and between entities must be fulfilled. Namely, we considered the modification of this problem studied and evaluated in [FRF+11a, ADV12], see the description of the test instances in Section 4.1.3. The benchmarks[1] correspond to a reconfiguration problem which includes knowledge describing a legacy solution (previous solution which has to be reconciled with new requirements) used for finding a reconfiguration solution. It is impossible to compare our results with the results provided in [FRF+11a] since the presence of a legacy solution in some cases simplifies the solving,

[1] http://proserver3-iwas.uni-klu.ac.at/reconcile/index.php/benchmarks

e.g. in the newroom problem instances. Ignoring the legacy knowledge we get a set of configuration problem instances which were evaluated using the plain ASP encoding without the rewriting and the rewriting program. Moreover, from four types of configuration problems described in [FRF+11a] we present the evaluation results for *Empty*, *Long* and *Newroom* instances. As it turned out during the experiments, *Swap* instances do not contain the binary conflicts necessary for an efficient application of the conflict-based rewriting and only the common encoding can be applied. The overhead for the rewriting of the hardest swap instance is about 3 minutes which is rather small in comparison to the overall solution time.

A knowledge base for a House problem instance comprises objects such as person, thing, cabinet, and room, where things can be long or short and cabinets high or small. The number of cabinets and rooms generated for each problem equals the number of things such that a fresh constant can be used in each relation. The configuration problem is to place these things into cabinets and the cabinets into rooms such that the requirements mentioned in Section 3.1 are satisfied. A solution with minimal number of high cabinets, cabinets and rooms is preferred.

The algorithms presented in Section 5.1 as well as supporting methods, such as finding of justified choice rules or their modifications, were implemented in JAVA. In our evaluation[2] we tested the algorithm allowing the backtracking only in a situation when a set partial solution cannot be extended. That is, any extension results in an unsatisfiable program due to non-binary constraints. The algorithm quits as soon as the first solution is found.

The set of benchmark problems comprised 24 configuration instances including 8 instances of each type. The name of an instance indicates the number of persons and things declared in an EDB. The size of the ground program depending on an EDB varies from 0.6 to 620 Mb with an average of 133 Mb (LPARSE format). The ASP program corresponding to the House problem included a set of rules expressing the requirements given above. The costs of a solution were determined as a number of generated components used in a solution.

We compared the suggested rewriting approach with an ASP program without rewriting referred to as *Rewriting* and *Original* respectively. We measured execution time and optimality of the solution found within 900 seconds. In both cases we used the ordering rules as defined in Section 5.1.

[2]The evaluation experiments were performed using the grounder GRINGO (v. 3.0.4) and the solver CLASP (v. 2.0.6) with default portfolio options on a system with Intel i7-3930K CPU (3.20GHz), 32GB of RAM and running Ubuntu 11.10. Note that the method is not only restricted to the mentioned ASP system. Usage of systems such as DLV [LPF+06] is possible with some modifications in the translation, since choice rules are not supported by its knowledge representation language.

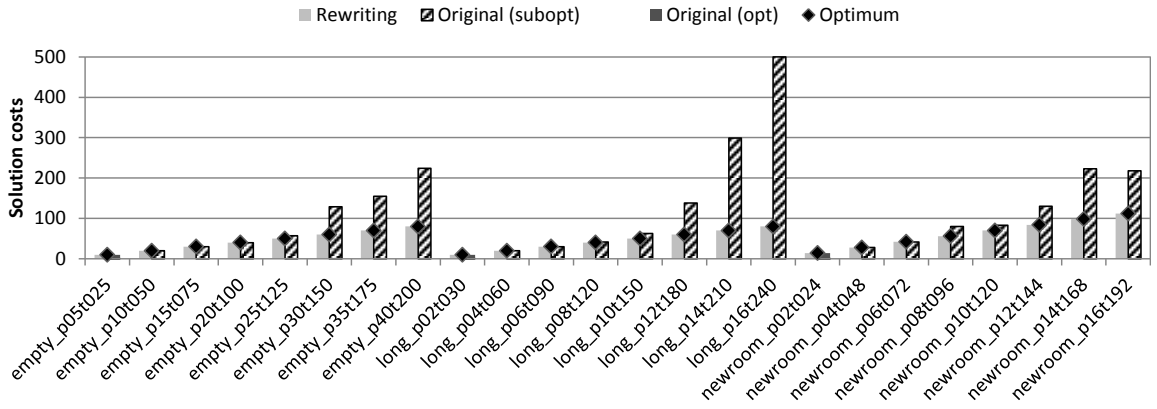

Figure 5.1: Quality of solutions identified by Original and Rewriting in 900 seconds

The evaluation showed that the solver Clasp was able to find solutions with the minimal costs, i.e. minimum number of used nulls (high cabinets/cabinets/rooms) for the 10 smallest cases presented in Fig. 5.1 using *Original*. Among them it was able to prove the optimality only for three smallest instances empty_p05t025, long_p02t030, newroom_p02t024 in 0.3, 1.2 and 103 seconds respectively which are presented by *Original (opt)*. Generally, the solver had a problem with proving optimality of a solution given *Original* encoding and in most of the cases the timeout was reached – *Original (subopt)*. In opposite, *Rewriting* was able to find a solution with optimal costs for all considered cases requiring 640 seconds for the hardest instance long_p16t240. Additionally, we did an experiment running the solver on the instance long_p16t240 for 3 hours with *Original* and the solver still failed to improve the costs of the preferred solution. Thus, *Rewriting* finds significantly better solutions in less time than *Original*.

Fig. 5.2 presents the overall runtime of *Rewriting* solving the House configuration problem including three optimization criteria, i.e. minimizing the number of high cabinets, cabinets and rooms. The total runtime was divided into the following processing stages: grounding, parsing, conflict detection, nulls generation, rewriting and solving. All *Empty*, *Long* and *Newroom* instances are grouped depending on the average number of things. We took the number of things to represent the instances and their average time to illustrate the results. Here we provide time segments for all instances and a total time using *Rewriting*. As the diagram shows, the hardest instance was solved in about 640 seconds including 225 seconds for grounding, 170 seconds for the conflict detection and 205 seconds for solving. The remaining 40 seconds were used for parsing, generation of nulls and rewriting. The rewriting takes very little time, 79 milliseconds for the mentioned test instance.

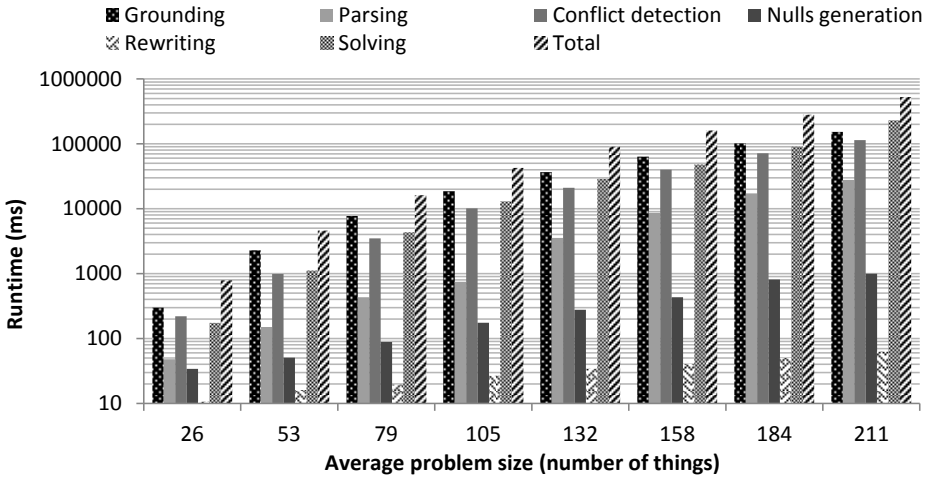

Figure 5.2: Time to find a solution using three optimization criteria and Rewriting

This makes it invisible on the diagram compared with other stages for the smallest test instances.

The evaluation results demonstrate the convincing superiority of the rewriting approach over the common (original) encoding using state-of-the art solvers for instances relevant to practice. Modifying the existential rules by elimination of the problems' binary conflicts allows to speed up computations for a typical class of configuration problems occurring in practice and strongly enhance the quality of returned solutions.

CHAPTER

6 Conclusions

The aim of this work was to develop a knowledge representation and reasoning (KRR) approach for both configuration and reconfiguration problems. The contributions presented in this thesis can be summarized as follows:

- research on configuration problems and practical reconfiguration scenarios;
- analysis of configuration systems as well as KRR approaches for (re)configuration of complex products and services;
- elaboration of a formal method for defining (re)configuration problems using a logic-based formalism;
- prototypical implementations of the proposed method using modern solving systems;
- evaluation of the methods implemented on the representative set of (re)configuration problem instances;
- development and evaluation of a decomposition method for improving the efficiency of reasoning and quality of found solutions.

To achieve our goal we analyzed existing knowledge-based approaches. We started from rule-based and case-based approaches and then specifically focused on model-based methods. The latter are successfully applied in many systems available on the market. In addition, we investigated different configuration problems and practical reconfiguration scenarios occurring in Siemens' practice and in the literature [ADF+11, FRF+11a, RPF+12, HFS+14]. As a result, we created a representative

set of benchmarks reflecting typical configuration and customer requirements. The practical knowledge about possible (re)configuration problems was used as a background for our language. This methodology ensures that the developed language is applicable to multiple practical configuration problems.

While developing the knowledge representation language we faced another challenge – the computational complexity of (re)configuration problems. This is mainly due to the number of components available in the catalog, the number of possible connections between them and the complexity of customer and system requirements. The first two factors result in a very large number of possible candidate solutions that must be explored by a (re)configuration system. The last factor often leads to situations where only a tiny fraction of candidate solutions are actually solutions to a (re)configuration problem.

Therefore, we investigated problem solving methods that proved their utility in similar domains. In particular, in our work we focused on two knowledge representation and reasoning formalisms, namely Answer Set Programming (ASP) and Constraint Programming (CP). For both formalisms there is a number of efficient solvers that can be used to find solutions of (re)configuration problems encoded in our language. For that reason, we developed a translation of our language into both ASP and CP in order to compare the performance of the solvers available for (re)configuration problems.

The evaluation results for the set of (re)configuration benchmarks showed an admissible efficiency [ADF+11, FRF+11a, RPF+12, Rya12, FSFR12, SFRF13]. Interestingly, standard ASP solvers have outperformed CSP solvers on average. However, the largest problem instances remained unsolved within a given time frame by either of the solvers used in the evaluation. To overcome this problem we suggested and evaluated some specific solving techniques including domain-specific heuristics, symmetry breaking approaches and solver tuning strategies.

In many cases these techniques were able to speed-up the computations, but the success on large-scale real-world instances was still limited. The sophisticated heuristics and symmetry breaking strategies created have performed in an unstable fashion, in several cases even making the solving process deteriorate. This is particularly the case for the optimization benchmarks for which almost no improvement was shown. The main problem of all applied solvers was to prove the optimality of the found solution. That is, in many cases the solver returned a solution with optimal costs, but was unable to check its optimality. This is due to a large number of non-solutions with lower costs being present in the search space. The latter are some invalid (re)configurations, e.g. because they use less

components than required, which are less costly than valid (re)configurations. Consequently, we had to reduce the number of irrelevant elements in the search space in order to achieve the required performance.

The decomposition method presented in this thesis [RFF13] reduces the number of possible configurations by excluding combinations of components that cannot occur in any solution. Moreover, these exclusions are implemented so that they also eliminate symmetric solutions from the search space. Additionally, we modified the reasoning process in a way that allows the problems to be solved level-by-level. That is, the solver is used to build a partial solution on one level. Then, the partial solution is assumed to be fixed on the next level and is only extended by a solver. The evaluation of the suggested decomposition method showed that it was able to find (optimal) solutions for problems occurring in practice within an acceptable time. Moreover, it was able to improve the quality of solutions for some classes of real-world configuration problems.

Finally, it is worth stating that, in general, finding (optimal) (re)configurations remains a challenge for realistic problem cases. The first principal difficulty for ASP-based approaches is the explosion of grounding. The large number of constants and atoms over non-unary predicates, typical for real-world problems, results in a very large number of ground atoms. Therefore, given a non-ground program an ASP grounder might output a ground program which is significantly larger than the input.

The identification of appropriate domain-specific heuristics for a particular problem using search-based approaches is the next crucial issue. The generation of heuristics for the problems studied in this work was not supported by any automatic approach. This can be seen as one of the initial motivations for designing methods which are able to automatically create heuristics that will guide the search of solutions in general problem solvers such as ASP or CP reasoners. Of course, in order to express these heuristics an appropriate heuristic representation language has to be found/developed and integrated in general-purpose frameworks.

Investigations in these two directions are already very encouraging and will be studied in our future work. Results of this research might significantly reduce the effort for generating domain-specific heuristics and the computation costs for finding (optimized) solutions, i.e. configurations, in practice. Consequently, more efficient production processes will be enabled and, thus, better configuration products and services can be supplied to the customers.

Bibliography

[ADF+11] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon, and E. Thorstensen. Optimization Methods for the Partner Units Problem. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 4–19, 2011.

[ADG+11] M. Aschinger, C. Drescher, G. Gottlob, P. Jeavons, and E. Thorstensen. Tackling the Partner Units Configuration Problem. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 497–503, 2011.

[ADV12] M. Aschinger, C. Drescher, and H. Vollmer. LoCo – A Logic for Configuration Problems. In *Proceedings of the European Conference on Artificial Intelligence*, pages 73–78, 2012.

[AFL+11] M. Alviano, W. Faber, N. Leone, S. Perri, G. Pfeifer, and G. Terracina. The disjunctive datalog system DLV. In *Proceedings of the International Workshop Datalog Reloaded*, pages 282–301, 2011.

[AGL+05] C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The nomore++ approach to answer set solving. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 95–109, 2005.

[AP94] A. Aamodt and E. Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59, 1994.

[ARMS03] F. A. Aloul, A. Ramani, I. L. Markov, and K. A Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[ASM06] F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient Symmetry Breaking for Boolean Satisfiability. *IEEE Transactions on computers*, 55(5):549–558, 2006.

[Bac88] J. Bachant. RIME: Preliminary work toward a knowledge-acquisition tool. In *Automating Knowledge Acquisition for Expert Systems*, pages 201–224. 1988.

[BCM+10] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2010.

[BET11] G. Brewka, T. Eiter, and M. Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BFK85] L. Brownston, R. Farrell, and E. Kant. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.

[BHS08] F. Baader, I. Horrocks, and U. Sattler. Description Logics. In *Handbook of Knowledge Representation*, pages 135–179. Elsevier Science, 2008.

[BPS13] P. Blazek, M. Partl, and C. Streichsbier. Configurator Database 2013. Technical report, CEO of cyLEDGE Media, 2013.

[CGK08] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 70–80, 2008.

[CGLP10] A. Calì, G. Gottlob, T. Lukasiewicz, and A. Pieris. Datalog+/-: A Family of Languages for Ontology Querying. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 351–368, 2010.

[CGS+89] R. Cunis, A. Günter, I. Syska, H. Peters, and H. Bode. Plakon - an approach to domain-independent construction. In *Proceedings of the IEA/AIE*, pages 866–874, 1989.

[CIR14] F. Calimeri, G. Ianni, and F. Ricca. The third open answer set programming competition. *Theory and Practice of Logic Programming*, 14(01):117–135, 2014.

[CKR09] D. Conry, Y. Koren, and N. Ramakrishnan. Recommender systems for the conference paper assignment problem. In *Proceedings of the ACM conference on Recommender systems*, pages 357–360, 2009.

[Cla78] K. L. Clark. Negation as failure. In *Logic and data bases*, pages 293–322, 1978.

[CR91] J. Crow and J. M. Rushby. Model-Based Reconfiguration: Toward an Integration with Diagnosis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 836–841, 1991.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DN92] S.T. Dumais and J. Nielsen. Automating the Assignment of Submitted Manuscripts to Reviewers. In *Proceedings of the Conference on Research and Development in Information Retrieval*, pages 233–244, 1992.

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[Dre12] C. Drescher. The Partner Units Problem a Constraint Programming Case Study. In *Proceedings of the International Conference on Tools with Artificial Intelligence*, pages 170–177, 2012.

[DSTW08] J. P. Delgrande, T. Schaub, H. Tompits, and S. Woltran. Belief Revision of Logic Programs under Answer Set Semantics. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 411–421, 2008.

[DTW11] C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking Answer Set Solving. *AI Communications*, 24(2):177–194, 2011.

[EFL+03] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLVK system. *Artificial Intelligence*, 144(1):157–211, 2003.

[EFLP99] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1):99–111, 1999.

[EIK09] T. Eiter, G. Ianni, and T. Krennwallner. Answer Set Programming: A Primer. In *Reasoning Web*, pages 40–110, 2009.

[Fab05] W. Faber. Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 40–52, 2005.

[Fei13] I. Feinerer. Efficient large-scale configuration via integer linear programming. *AI for Engineering Design, Analysis and Manufacturing*, 27:37–49, 2013.

[FFH+98] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring Large Systems Using Generative Constraint Satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.

[FFJ00] A. Felfernig, G. Friedrich, and D. Jannach. UML as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering*, 10(4):449–469, 2000.

[FFJS04] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234, 2004.

[FFM+10] G. Friedrich, M. Fugini, E. Mussi, B. Pernici, and G. Tagni. Exception Handling for Repair in Service-Based Processes. *IEEE Transactions on Software Engineering*, 36(2):198–215, 2010.

[FFSS10] A. Falkner, I. Feinerer, G. Salzer, and G. Schenner. Computing Product Configurations via UML and Integer Linear Programming. *Journal of Mass Customisation*, 3(4):351–367, 2010.

[FH13] A. Falkner and A. Haselböck. Challenges of Knowledge Evolution in Practice. *AI Communications*, 26:3–14, 2013.

[FHBT14] A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, editors. *Knowledge-based Configuration: From Research to Business Cases*. Morgan Kaufmann, 2014.

[FHSS11] A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner. Modeling and Solving Technical Product Configuration Problems. *AI EDAM*, 25(2):115–129, 2011.

[FKP05] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: getting to the core. *ACM Transactions on Database Systems*, 30(1):174–210, 2005.

[FLP12] W. Faber, N. Leone, and S. Perri. The intelligent grounder of DLV. In *Correct Reasoning*, pages 247–264, 2012.

[FMK+08] G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou. Ontology change: classification and survey. *The Knowledge Engineering Review*, 23(2):117–152, 2008.

[Fre91] E. C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 227–233, 1991.

[FRF+11a] G. Friedrich, A. Ryabokon, A. A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner. (Re) configuration based on model generation. In *Proceedings of the International Workshop on Logics for Component Configuration*, pages 26–35, 2011.

[FRF+11b] G. Friedrich, A. Ryabokon, A.A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner. (Re)configuration using Answer Set Programming. In *Proceedings of the IJCAI Workshop on Configuration*, pages 17–25, 2011.

[FS14] A. Falkner and H. Schreiner. SIEMENS: Configuration and Reconfiguration in Industry. *Klowledge-Based Configuration: From Research to Business Cases*, pages 199–210, 2014.

[FSFR12] A. Falkner, G. Schenner, G. Friedrich, and A. Ryabokon. Testing Object-Oriented Configurators With ASP. In *Proceedings of the ECAI Workshop on Configuration*, pages 21–26, 2012.

[FSG+09] P. A. Flach, S. Spiegler, B. Golénia, S. Price, J. Guiver, R. Herbrich, T. Graepel, and M. J. Zaki. Novel tools to streamline the conference review process: experiences from SIGKDD'09. *SIGKDD Explorations*, 11(2):63–67, 2009.

[GGI+10] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and Prediction (under Inconsistency) in Large Biological Networks with Answer Set Programming. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 497–507, 2010.

[GHK02] I. P. Gent, W. Harvey, and T. Kelsey. Groups and Constraints: Symmetry Breaking during Search. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, pages 415–430, 2002.

[GK99] A. Günter and C. Kühn. Knowledge-Based Configuration – Survey and Future Directions. In *Proceedings of the German Conference on Knowledge Based Systems*, Lecture Notes in Artificial Intelligence, 1999.

[GKK+08] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an Incremental ASP Solver. In *Proceedings of the International Conference on Logic Programming*, pages 190–205, 2008.

[GKK+10] N. Garg, T. Kavitha, A. Kumar, K. Mehlhorn, and J. Mestre. Assigning Papers to Referees. *Algorithmica*, 58(1):119–136, 2010.

[GKK+11a] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A Portfolio Solver for Answer Set Programming: Preliminary Report. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 352–357, 2011.

[GKK+11b] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T. Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, 24(2):107–124, 2011.

[GKKS11] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in gringo series 3. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 345–351, 2011.

[GKKS12] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[GKNS07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 136–148, 2007.

[GKO+09] M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub, and S. Thiele. On the Input Language of ASP Grounder Gringo. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 502–508, 2009.

[GKS12] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–386, 1991.

[GLL+04] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.

[GLM06] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[GN07] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[GS07] J. Goldsmith and R.H. Sloan. The AI conference paper assignment problem. In *Proceedings of the AAAI Workshop on Preference Handling for Artificial Intelligence*, pages 53–57, 2007.

[Gün95] A. Günter. Konwerk - ein modulares Konfigurierungswerkzeug. *Expertensysteme*, 95:1–18, 1995.

[Has93] A. Haselböck. Exploiting Interchangeabilities in Constraint-Satisfaction Problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 282–289, 1993.

[HB11] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool, 2011.

[HFS+14] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, and K. Wolter. Configuration Knowledge Representation and Reasoning. *Klowledge-Based Configuration: From Research to Business Cases*, pages 41–72, 2014.

[HJ91] M. Heinrich and E. Jungst. A Resource-Based Paradigm for the Configuring of technical Systems from Modular Components. In *IEEE Conference on Artificial Intelligence Applications*, pages 257–264, 1991.

[HS14] A. Haselböck and G. Schenner. S'UPREME. *Klowledge-Based Configuration: From Research to Business Cases*, pages 263–269, 2014.

[HWC99] D. Hartvigsen, J.C. Wei, and R. Czuchlewski. The Conference Paper-Reviewer Assignment Problem*. *Decision Sciences*, 30(3):865–876, 1999.

[JK07] T. Junttila and P. Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. *Proceedings of the Workshop on Algorithm Engineering and Experiments*, pages 135–149, 2007.

[Jun06] U. Junker. Configuration. *Handbook of Constraint Programming*, 2:837–874, 2006.

[JZFF10] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich. *Recommender Systems: An Introduction*. Cambridge University Press, 2010.

[KR99] I. Kreuz and D. Roller. Knowledge growing old in reconfiguration context. *Proceedings of the AAAI Workshop on Configuration*, 1999.

[LMTV12] N. Leone, M. Manna, G. Terracina, and P. Veltri. Efficiently Computable Datalog∃ Programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 13–23, 2012.

[LPF+06] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)*, 7(3):499–562, 2006.

[LRS97] N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics, and Computation. *Information and Computation*, 135(2):69–112, 1997.

[LZ04] F. Lin and Y. Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1):115–137, 2004.

[LZH06] Z. Lin, Y. Zhang, and H. Hernandez. Fast SAT-based answer set solver. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 92–97, 2006.

[Man05] P. Manhart. Reconfiguration - A problem in search of solutions. In *Proceedings of the IJCAI Configuration Workshop*, pages 64–67, 2005.

[MBSF09] W. Mayer, M. Bettex, M. Stumptner, and A. Falkner. On solving complex rack configuration problems using CSP methods. *Proceedings of the IJCAI Workshop on Configuration*, 2009.

[McD82] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[McG10] D. L. McGuinness. Configuration. In *Description Logic Handbook*, pages 388–405. Cambridge University Press, 2010.

[MF89] S. Mittal and F. Frayman. Towards a generic model of configuraton tasks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1395–1401, 1989.

[MF90] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the AAAI*, pages 25–32, 1990.

[MII+02] D.F. Manlove, R.W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. *Theoretical Computer Science*, 276(1-2):261–279, 2002.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the Design Automation Conference*, pages 530–535, 2001.

[MSM88] S. Marcus, J. Stout, and J. McDermott. VT: An Expert Elevator Designer that uses Knowledge-based Backtracking. *AI Magazine*, pages 95–112, 1988.

[MSTS99] T. Männistö, T. Soininen, J. Tiihonen, and R. Sulonen. Framework and conceptual model for reconfiguration. In *Proceedings of the AAAI Workshop on Configuration*, pages 59–64, 1999.

[MW98] D. McGuinness and J. R. Wright. Conceptual modelling for configuration: A Description Logic-based approach. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 12:333–344, 1998.

[NBG+01] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A prolog decision support system for the Space Shuttle. In *Proceedings of the International Answer Set Programming Workshop*, 2001.

[OFSA11] M. Ostrowski, G. Flouris, T. Schaub, and G. Antoniou. Evolution of Ontologies using ASP. *Proceedings of the International Conference on Logic Programming (Technical communications)*, pages 16–27, 2011.

[OK88] B. Owsnicki-Klewe. Configuration as a consistency maintenance task. In *Proceedings of the German Workshop on Artificial Intelligence*, pages 77–87, 1988.

[PS08] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, World Wide Web Consortium, 2008.

[Rei87] R. Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[RFF13] A. Ryabokon, G. Friedrich, and A. A. Falkner. Conflict-Based Program Rewriting for Solving Configuration Problems. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 465–478, 2013.

[RGA+12] F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, and N. Leone. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming*, 12(3):361–381, 2012.

[RPF+12] A. Ryabokon, A. Polleres, G. Friedrich, A. A. Falkner, A. Haselböck, and H. Schreiner. (Re)Configuration Using Web Data: A Case Study on the Reviewer Assignment Problem. In *Proceedings of the Web Reasoning and Rule Systems*, pages 258–261, 2012.

[RS08] V. Ryvchin and O. Strichman. Local Restarts. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 271–276, 2008.

[RV96] J. Rahmer and A. Voß. Case-enhanced configuration by resource balancing. In *Proceedings of the Workshop on Advances in Case-Based Reasoning*, pages 339–353, 1996.

[Rya12] A. Ryabokon. Study: Symmetry breaking for ASP. *CoRR*, abs/1212.2:1–11, 2012.

[Ryc96] N. Rychtyckyj. DLMS: An evaluation of KL-ONE in the automobile industry. In *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning*, pages 588–596, 1996.

[Sak09] K. A. Sakallah. Handbook of Satisfability. In *Handbook of Satisfiability*, pages 289–338. IOS Press, 2009.

[SBF01] G. D. Silveira, D. Borenstein, and H. S. Fogliatto. Mass customization : Literature review and research directions. *International Journal of Production Economics*, 72(1):1–13, 2001.

[SFH98] M. Stumptner, G. Friedrich, and A. Haselböck. Generative constraint-based configuration of large technical systems. *AI EDAM*, 12(4):307–320, 1998.

[SFRF13] G. Schenner, A. Falkner, A. Ryabokon, and G. Friedrich. Solving Object-oriented Configuration Scenarios with ASP. In *Proceedings of the International Configuration Workshop*, pages 55–62, 2013.

[SH07] C. Sinz and A. Haag. Configuration. *IEEE Intelligent Systems*, 22(1):78–90, 2007.

[SHF94] M. Stumptner, A. Haselböck, and G. Friedrich. COCOS – A tool for constraint-based, dynamic configuration. In *Artificial Intelligence for Applications*, pages 373–380, 1994.

[SL10] M. Slota and J. Leite. On Semantic Update Operators for Answer-Set Programs. In *Proceedings of the European Conference on Artificial Intelligence*, pages 957–962, 2010.

[SMSS03] L. Stojanovic, A. Maedche, N. Stojanovic, and R. Studer. Ontology evolution as reconfiguration-design problem solving. In *Proceedings of the International Conference on Knowledge Capture*, pages 162–171, 2003.

[SN01] T. Syrjänen and I. Niemelä. The Smodels system. In *Proceedings of the International Conference Logic Programming and Nonmonotonic Reasoning*, pages 434–438, 2001.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[SNTS01] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the International Workshop on Answer Set Programming*, pages 195–201, 2001.

[Str08] P. Struss. Model-based problem solving. *Handbook of Knowledge Representation*, pages 395–465, 2008.

[Stu97] M. Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2):111–125, 1997.

[SW98a] D. Sabin and R. Weigel. Product configuration frameworks - a survey. *IEEE Intelligent Systems (EXPERT)*, 13(4):42–49, 1998.

[SW98b] M. Stumptner and F. Wotawa. Model-based reconfiguration. In *Proceedings of the International Conference on Artificial Intelligence in Design*, pages 45–64, 1998.

[TCC05] H.-E. Tseng, C.-C. Chang, and S.-H. Chang. Applying case-based reasoning for product configuration in mass customization environments. *Expert Systems with Applications*, 29(4):913–925, 2005.

[TFF12] E. C. Teppan, G. Friedrich, and A. A. Falkner. QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem. In *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, pages 2329–2334, 2012.

[THAS13] J. Tiihonen, M. Heiskala, A. Anderson, and T. Soininen. WeCoTin - A practical logic-based sales configurator. *AI Communications*, 26(1):99–131, 2013.

[Wal06] Toby Walsh. General symmetry breaking constraints. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, pages 650–664, 2006.

[WWV+93a] J. R. Wright, E. S. Weixelbaum, G. T. Vesonder, K. E. Brown, S. R. Palmer, J. I. Berman, and H. H. Moore. A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems. *AI Magazine*, 14(3):69–80, 1993.

[WWV+93b] J.R. Wright, E. S. Weixelbaum, G. T. Vesonder, K. E. Brown, S. R. Palmer, J. I. Berman, and H. H. MooreRoman. A Knowledge-Based Configurator That Supports Sales, Engineering, and Manufacturing at AT&T Network Systems. In *Proceedings of the Innovative Applications of Artificial Intelligence Conference*, pages 183–193, 1993.

[YF99] D. Yarowsky and R. Florian. Taking the Load Off the Conference Chairs: Towards a Digital Paper-Routing Assistant. In *Proceedings of the Joint SIGDAT Conference on Empirical Methods in NLP and Very-Large Corpora*, pages 90–99, 1999.

Appendices

APPENDIX

A *Pearl heuristic* for the PUP (ASP)

```
% INITIAL PROGRAM, i.e. configuration requirements
zone(Z) :- zone2sensor(Z,D).
doorSensor(D) :- zone2sensor(Z,D).

comUnit(1..lower).

1 {unit2zone(U,Z) : comUnit(U)} 1 :- zone(Z).
:- comUnit(U), 3 {unit2zone(U,Z) : zone(Z)}.

1 {unit2sensor(U,D) : comUnit(U)} 1 :- doorSensor(D).
:- comUnit(U), 3 {unit2sensor(U,D) : doorSensor(D)}.

partnerunits(U,P) :- unit2zone(U,Z), zone2sensor(Z,D), unit2sensor(P,D), U!= P.
partnerunits(U,P) :- partnerunits(P,U), comUnit(U), comUnit(P).
:- comUnit(U), maxPU+1 {partnerunits(U,P) : comUnit(P)}.

% Ring heuristic
partners(U1,U2) :- comUnit(U1), comUnit(U2), U1=U2-1.
partnerunits(P,U) :- partners(P,U).
partnerunits(U,P) :- partners(U,P).
```

```
% PEARL HEURISTIC
% ASSIGNMENT OF ZONES
% assign first zone to the first unit
firstZone(Y) :- Y = #min[zone(X)=X].
firstUnit(Y) :- Y = #min[comUnit(X)=X].
unit2zone(U,Z) :- firstZone(Z), firstUnit(U).

% define adjacent zones
adjacent(Z1,D,Z2) :- zone2sensor(Z1,D), zone2sensor(Z2,D), Z1<Z2.
% define a column
column(Z1,D,Z2) :- zone(Z1), zone(Z2), adjacent(Z1,D,Z2), #abs(Z2−Z1) > 1 .
% assign column on one unit
unit2zone(U,Z1) :- column(Z,_,Z1), unit2zone(U,Z).
% assign next column on the next unit
unit2zone(P,Z2) :- unit2zone(U,Z), zone(Z2), Z2=Z+1, P #mod 3 > 0, partners(U,P).
% ev ery third unit should be free
unit2zone(X,Z2) :- unit2zone(U,Z), zone(Z2), Z2=Z+1, P #mod 3 == 0,
                   partners(U,P), partners(P,X).

% ASSIGNMENT OF DOOR SENSORS
ev en(Z) :- column(_,_,Z), firstZone(ZF), column(ZF,_,Z3), (Z−Z3) #mod 2 == 0.
odd(Z) :- column(_,_,Z), firstZone(ZF), column(ZF,_,Z3), (Z−Z3) #mod 2 > 0.
rightSensor(Z,D) :- adjacent(Z,D,Z1), Z1=Z+1.

% assign inner column doors
unit2sensor(U,D) :- column(Z,D,_), unit2zone(U,Z).
% assign top row : column is used to distinguish the rows
unit2sensor(P,D) :- column(Z,_,_), rightSensor(Z,D), unit2zone(U,Z), partners(U,P).
% assign bottom row : difference between the first zone in the row and the current
unit2sensor(U,D) :- rightSensor(Z,D), unit2zone(U,Z), ev en(Z).
unit2sensor(P,D) :- rightSensor(Z,D), unit2zone(U,Z), odd(Z), partners(U,P).
```

APPENDIX

B House problem (ASP)

B.1 *Original* encoding

```
% INITIAL PROBLEM
cabinetDomain(X) :- cabinetDomainNew(X).
roomDomain(X) :- roomDomainNew(X).

% definition of lower and upper numbers of cabinets and rooms possible in a configuration
cabinetLower {cabinet(X) : cabinetDomain(X)} cabinetUpper.
roomLower {room(X) : roomDomain(X)} roomUpper.

% ordering of used cabinets and rooms
room(X) :- roomDomainNew(X), roomDomainNew(Y), room(Y), X<Y.
cabinet(X) :- cabinetDomainNew(X), cabinetDomainNew(Y), cabinet(Y), X<Y.

% association cabinetTOthing
1 {cabinetTOthing(X,Y) : cabinetDomain(X)} 1 :- thing(Y).
:- 6 {cabinetTOthing(X,Y) : thing(Y)}, cabinet(X).

% association roomTOcabinet
1 {roomTOcabinet(X,Y) : roomDomain(X)}1 :- cabinet(Y).
:- 5 {roomTOcabinet(X,Y) : cabinetDomain(Y)}, room(X).
```

```
% association personTOroom
% a room belongs to a person, who stores things in cabinets in that room
personTOroom(P,R) :- personTOthing(P,X), cabinetTOthing(C,X), roomTOcabinet(R,C).

% things of one person cannot be placed in a cabinet together with things of another person
:- cabinetTOthing(X,Y1), cabinetTOthing(X,Y2),
                personTOthing(P1,Y1), personTOthing(P2,Y2), P1!= P2.

% cabinets of different people cannot be placed in the same room
% i.e. one and the same room cannot belong to 2 different persons
:- personTOroom(P1,R), personTOroom(P2,R), P1!= P2.

room(X) :- roomTOcabinet(X,Y).
room(Y) :- personTOroom(X,Y).

cabinet(X) :- cabinetTOthing(X,Y).
cabinet(Y) :- roomTOcabinet(X,Y).

% MODIFIED PROBLEM
% there are 2 types of things which are disjoint
thing(X) :- thingLong(X).
thing(X) :- thingShort(X).
:- thingLong(X), thingShort(X).

% long things hav e to be packed in high cabinets
1 {cabinetHigh(X), cabinetSmall(X)} 1 :- cabinet(X).
cabinetHigh(C) :- thingLong(X), cabinetTOthing(C,X).

% at most either 2 high things or 1 high and 2 short or 4 short cabinets
% are allowed to be in a room
```

```
% assumption : there are 4 cabinet slots in each room.
%High cabinet requires 2 slots, small − 1 slot
cabinetSize(X,1) :- cabinet(X), cabinetSmall(X).
cabinetSize(X,2) :- cabinet(X), cabinetHigh(X).
roomTOcabinetSlot(R,C,S) :- roomTOcabinet(R,C), cabinetSize(C,S).
:- 5 [roomTOcabinetSlot(X,Y,S) : cabinetDomain(Y)=S], room(X).

% TRANSFORMATION RULES
person(X) :- legacyConfig(person(X)).
thing(X) :- legacyConfig(thing(X)).
personTOthing(X,Y) :- legacyConfig(personTOthing(X,Y)).

1 {reuse(cabinetTOthing(X,Y)), delete(cabinetTOthing(X,Y))} 1 :-
        legacyConfig(cabinetTOthing(X,Y)).
1 {reuse(roomTOcabinet(X,Y)), delete(roomTOcabinet(X,Y))} 1 :-
        legacyConfig(roomTOcabinet(X,Y)).
1 {reuse(personTOroom(X,Y)), delete(personTOroom(X,Y))} 1 :-
        legacyConfig(personTOroom(X,Y)).

% all cabinets from the original solution are added to the new domain
cabinetDomain(X) :- legacyConfig(cabinet(X)).
1 {reuse(cabinet(X)), delete(cabinet(X))} 1 :- legacyConfig(cabinet(X)).

% all cabinets from the original solution are added to the new domain
roomDomain(X) :- legacyConfig(room(X)).
1 {reuse(room(X)), delete(room(X))} 1 :- legacyConfig(room(X)).

% all reused atoms should be a part of a reconfiguration
% i.e. they should be defined as facts
cabinetTOthing(X,Y) :- reuse(cabinetTOthing(X,Y)).
roomTOcabinet(X,Y) :- reuse(roomTOcabinet(X,Y)).
```

```
personTOroom(X,Y) :- reuse(personTOroom(X,Y)).
cabinet(X) :- reuse(cabinet(X)).
room(X) :- reuse(room(X)).

% all deleted atoms cannot be used in a configuration
:- cabinetTOthing(X,Y), delete(cabinetTOthing(X,Y)).
:- roomTOcabinet(X,Y), delete(roomTOcabinet(X,Y)).
:- personTOroom(X,Y), delete(personTOroom(X,Y)).
:- cabinet(X), delete(cabinet(X)).
:- room(X), delete(room(X)).

% COSTS
% Creation costs
cost(create(cabinetHigh(X)),W) :- cabinetHigh(X), cabinetHighCost(W),
                                  cabinetDomainNew(X).
cost(create(cabinetSmall(X)),W) :- cabinetSmall(X), cabinetSmallCost(W),
                                   cabinetDomainNew(X).
cost(create(room(X)),W) :- room(X), roomCost(W), roomDomainNew(X).

% Reuse costs are defined similarly to the creation costs

% Deletion costs are defined similarly to the creation costs

% OPTIMIZATION
% Minimize reconfiguration costs
#minimize[cost(X,W) = W].
```

B.2 *Per person* heuristic

```
thing(X) :- legacyConfig(thing(X)).
person(X) :- legacyConfig(person(X)).
personTOthing(P,T) :- legacyConfig(personTOthing(P,T)).

% legacy cabinets/rooms of a person
cabinetL(P,L) :- person(P), L = {legacyConfig(cabinet(C)) :
                 legacyConfig(cabinetTOthing(C,T)) : legacyConfig(personTOthing(P,T))}.
roomL(P,L) :- person(P), L = {legacyConfig(room(X)) : legacyConfig(roomTOcabinet(X,C)) :
                 legacyConfig(cabinetTOthing(C,T)) : legacyConfig(personTOthing(P,T))}.

% a number of cabinets/rooms required per person
personCabinet(P,Y) :- person(P), nCabinetSmall(X), nCabinetHigh(Z), Y=X+Z.
personRoom(P,Y) :- person(P), nRoom(Y).

% S - a number of cabinets/rooms for the previous persons, everyone < current
cabinetD(500+S..500+S+O-L-1,P) :- person(P), S =[personCabinet(P1,C)=C : P1<P],
                                  personCabinet(P,O), cabinetL(P,L).
roomD(1000+S..1000+S+O-L-1,P) :- person(P), S=[personRoom(P1,C)=C : P1<P],
                                  personRoom(P,O), roomL(P,L).

% assign things of a person in cabinets of a person
1 {cabinetTOthing(X,T) : cabinetD(X,P) : personTOthing(P,T)} 1 :- thing(T).
:- 6 {cabinetTOthing(X,T) : thing(T)}, cabinetD(X,P).

% assign cabinets in rooms of a person
1 {roomTOcabinet(X,Y) : roomD(X,P) : personTOthing(P,T)} 1 :- cabinetTOthing(Y,T).
:- 5 {roomTOcabinet(X,Y) : cabinetD(Y,P)}, roomD(X,P).

personTOroom(P,R) :- roomTOcabinet(R,_), roomD(R,P).
```

room(*X*) :- *roomTOcabinet*(*X*,*Y*).

room(*Y*) :- *personTOroom*(*X*,*Y*).

cabinet(*X*) :- *cabinetTOthing*(*X*,*Y*).

cabinet(*Y*) :- *roomTOcabinet*(*X*,*Y*).

APPENDIX

C House problem (CSP)

```
include "globals.mzn";
% CONFIGURATION
% capacity of a cabinet
int: cabinetCap = 5;
% capacity of a room
int: roomCap = 4;

array [Things] of Persons : personTOthing;
set of int: PersonsDomain;
set of int: ThingsDomain;
set of int: CabinetsDomain;
set of int: RoomsDomain;

% for convinience...enumeration of set elements
set of int: Persons = 1..card(PersonsDomain);
set of int: Things = 1..card(ThingsDomain);
set of int: Cabinets = 1..card(CabinetsDomain);
set of int: Rooms = 1..card(RoomsDomain);

% an array of length |Things| including variables with domain Cabinets
array [Things] of var Cabinets : cabinetTOthing;
```

```
array [Things] of var Rooms : roomTOthing;

array[Cabinets] of int : cabinetLower = [0 | i in Cabinets];
array[Cabinets] of int : cabinetUpper = [cabinetCap | i in Cabinets];

% each cabinet can contain at most 5 things, built−in global constraint
% each value in the cabinetTOthing array should belong to the set Cabinets(i.e. closed)
% [i | i in Cabinets] is used to convert a set into an array as required by the global constraint
% each element of Cabinets can be used at least cabinetLower[i] and
% at most cabinetUpper[i] times
constraint global_cardinality_low_up_closed(cabinetTOthing, [i | i in Cabinets], cabinetLower,
    cabinetUpper);

% capacity contraints for rooms
array[Rooms] of int : roomLower = [0 | i in Rooms];
array[Rooms] of int : roomUpper = [roomCap*cabinetCap | i in Rooms];
constraint global_cardinality_low_up_closed(roomTOthing, [i | i in Rooms], roomLower,
    roomUpper);

array [Cabinets] of var Rooms : roomTOcabinet;
array[Rooms] of int : roomCabinetUpper = [roomCap | i in Rooms];
constraint global_cardinality_low_up_closed(roomTOcabinet, [i | i in Rooms], roomLower,
    roomCabinetUpper);

% if a thing is stored in a cabinet and in a room, then the cabinet is placed in the room
% i.e. if a thing is in a cabinet, and in a room, then a room of a cabinet is equal
% to value of the variable room
constraint forall (i in Things) (
 let {var int: cabinet = cabinetTOthing[i], var int: room = roomTOthing[i]} in roomTOcabinet[
     cabinet] = room);
```

```
% if 2 different things are placed in the same cabinet, they have to be owned by the same person
constraint forall (i,j in Things where i != j /\ personTOthing[i] != personTOthing[j]) (
        cabinetTOthing[i] != cabinetTOthing[j]);

% if 2 different things are placed in the same room they have to be owned by the same person
constraint forall (i,j in Things where i != j /\ personTOthing[i] != personTOthing[j]) (
        roomTOthing[i] != roomTOthing[j]);

% RECONFIGURATION
% input
set of int: LegacyCabinetsDomain;
set of int: LegacyRoomsDomain;

% this set enumerates cabinets in the legacy configuration
set of int: LegacyCabinets = 1..card(LegacyCabinetsDomain);

% this set enumerates rooms in the legacy configuration
set of int: LegacyRooms = 1..card(LegacyRoomsDomain);

set of int: BoolInt = {0,1};
% input: array of int − number of a room in which the thing was stored in the legacy configuration
array [Things] of int : legacyRoomTOthing;

% array of int which contains actual numbers of cabinets that the things are put to in the legacy
    configuration
array [Things] of int : legacyCabinetTOthing;

 % 0 corresponds to short and 1 to long things
array [Things] of BoolInt : thingLength;

% 0 corresponds to small and 1 to high cabinets
```

```
array [LegacyCabinets] of BoolInt : legacyCabinetHeight;

% decision variables
% use − true; delete − false
array [Cabinets] of var bool : useCabinet;
array [Rooms] of var bool : useRoom;

% 0 − small, 1 − high
array [Cabinets] of var BoolInt : cabinetHeight;

% useCabinet array of decision variables if true the cabinet is used, otherwise deleted
constraint forall (t in Things) (useCabinet[cabinetTOthing[t]] = true);
constraint forall (t in Things) (useRoom[roomTOthing[t]] = true);

% at least the same number of cabinets and rooms should be used in a reconfiguration
% as in a legacy configuration (no things are deleted)
constraint sum(i in Cabinets)(bool2int(useCabinet[i])) >= card(LegacyCabinetsDomain);
constraint sum(i in Rooms)(bool2int(useRoom[i])) >= card(LegacyRoomsDomain);

% if we delete a cabinet
% constraint (useCabinet[1] = false );

% if thing i is long, i.e. thingLength[i] = 1, then each cabinet c where this thing is stored
% i.e. cabinetTOthing[i] == CabinetsDomain[c] should be high, i.e. cabinetHeight[c] = 1

constraint forall (i in Things where thingLength[i] == 1)(
                    cabinetHeight[cabinetTOthing[i]] = 1);

% each room contains at most 4 slots that can be occupied by cabinets
% cabinetHeight is array of int (BoolInt) where 1 corresponds to high and 0 to small cabinet
% every high cabinet c1 occupies cabinetHeight[c1] + 1 = 1+1 = 2 slots
```

```
% every small cabinet c2 occupies cabinetHeight[c2] + 1 = 0+1 = 1 slot
% built−in function bool2int converts true to 1 and false to 0

% if cabinet is in the room then its slots are taken into account in the sum
% multipliers bool2int(isCabinetInRoom(c,r)) =1 for each cabinet placed in the room
% and =0 for each cabinet not in the room
constraint forall (r in Rooms) (sum(c in Cabinets)(
                bool2int(isCabinetInRoom(c,r))*(cabinetHeight[c]+1)) <= roomCap);

% predicate returns true if there exists a thing t that is stored in the room with the index roomInd
% and in the cabinet with the index cabInd
predicate isCabinetInRoom(int: cab, int: room) = exists(t in Things)(
        roomTOthing[t] == room /\ cabinetTOthing[t] == cab);

% COST: create costs
int: cabinetHighCost;
var int: cCabHigh = sum (c in Cabinets where (c in Cabinets diff LegacyCabinets))(
        cabinetHeight[c]*cabinetHighCost*bool2int(useCabinet[c]));

int: cabinetSmallCost;
var int: cCabSmall = sum (c in Cabinets where (c in Cabinets diff LegacyCabinets))(
        −1*(cabinetHeight[c]−1)*cabinetSmallCost*bool2int(useCabinet[c]));

int: roomCost;
var int: cRoom = sum (r in Rooms where (r in Rooms diff LegacyRooms))(
        roomCost*bool2int(useRoom[r]));

% compute overal costs (create; reuse, and delete costs analogously )
var int: overallCosts = cCabHigh + cCabSmall + cRoom;
constraint overallCosts >= 0;
```

```
% OPTIMIZATION
% uses the search annotation defined in *.dzn
ann: search_ann;

% otherwise a more general annotation can be used, e.g.
%search_ann = seq_search([int_search(cabinetTOthing, first_fail, indomain_min, complete),
    int_search(roomTOthing, first_fail, indomain_min, complete), int_search(roomTOcabinet,
    first_fail, indomain_min, complete)]);

solve
:: search_ann
%satisfy;
minimize overallCosts;

% OUTPUT
output ["thing(" ++ show(ThingsDomain[i]) ++ "," ++ if thingLength[i]==1 then "long" else "
    short" endif ++ ",
 " ++ show(CabinetsDomain[cabinetTOthing[i]]) ++ "," ++ show(RoomsDomain[roomTOthing[i
    ]]) ++ ",
 " ++show(PersonsDomain[personTOthing[i]]) ++ "). \n" | i in Things]

% cabinets output
        ++ ["cabinet(" ++ show(CabinetsDomain[i]) ++ "," ++ if fix(cabinetHeight[i]) == 1
            then "high"
  else "small" endif ++ "," ++
                if (i in LegacyCabinetsDomain) then
                        if fix(useCabinet[i]) then "reuse" else "delete" endif
                else
                        if fix(useCabinet[i]) then "created" else "notused" endif
                endif
                ++
```

```
        if fix(useCabinet[i]) then "," ++ show(RoomsDomain[roomTOcabinet[i]]) else
            "" endif
        ++ ").\n" | i in Cabinets]

% rooms output
    ++ ["room(" ++ show(RoomsDomain[i]) ++ "," ++
        if (i in LegacyRoomsDomain) then
            if fix(useRoom[i]) then "reuse" else "delete" endif
        else
            if fix(useRoom[i]) then "created" else "notused" endif
        endif
        ++ ").\n" | i in Rooms]

% costs output
    ++ ["cabinetHighCost(" ++ show(cCabHigh) ++ ").\n"]
    ++ ["cabinetSmallCost(" ++ show(cCabSmall) ++ ").\n"]
    ++ ["roomCost(" ++ show(cRoom) ++ ").\n"]
    ++ ["overallCosts(" ++ show(overallCosts) ++ ").\n"];
```