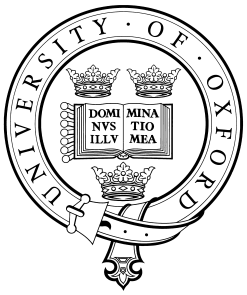
LoCo - A Logic for Configuration Problems



Markus Aschinger

St Anne's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary Term 2014

"There is something to be learned from a rainstorm. When meeting with a sudden shower, you try not to get wet and run quickly along the road. By doing such things as passing under the eaves of houses, you still get wet. When you are resolved from the beginning, you will not be perplexed, though you will still get the same soaking. This understanding extends to all things."

– Hagakure, the Way of the Samurai

LoCo - A Logic for Configuration Problems

Markus Aschinger

St Anne's College, University of Oxford

Doctor of Philosophy, Hilary Term 2014

Abstract

This thesis deals with the problem of technical product configuration: Connect individual components conforming to a component catalogue in order to meet a given objective while respecting certain constraints. Solving such configuration problems is one of the major success stories of applied AI research: In industrial environments they support the configuration of complex products and, compared to manual processes, help to reduce error rates and increase throughput. Practical applications are nowadays ubiquitous and range from configurable cars to the configuration of telephone communication switching units.

In the classical definition of a configuration problem the number of components to be used is fixed while in practice, however, the number of components needed is often not easily stated beforehand. Existing knowledge representation (KR) formalisms expressive enough to deal with this dynamic aspect of configuration require that explicit bounds on all generated components are given as well as extensive knowledge about the underlying solving algorithms. To date there is still a lack of high-level KR tools being able to cope with these demands.

In this work we present LoCo, a fragment of classical first order logic that has been carefully tailored for expressing technical product configuration problems. The core feature of LoCo is that the number of components used in configurations does not have to be finitely bounded explicitly, but instead is bounded implicitly through the axioms. We identify configurations with models of the logic; hence, configuration finding becomes model finding. LoCo serves as a high-level representation language which allows the modelling of general configuration problems in an intuitive and declarative way without the need of having knowledge about underlying solving algorithms; in fact, the specification gets automatically translated into low-level executable code. LoCo allows translations into different target languages. We present the language, related algorithms and complexity results as well as a prototypical implementation via answer-set programming.

Acknowledgements

First and foremost, I would like to thank my primary supervisor Georg Gottlob for giving me the opportunity to work on this project and for continuously pushing me to get things done. Second, I would like to thank Conrad Drescher who has worked as a PostDoc in the same project and with whom I have been sharing an office throughout my studies. His continuous support and the collaboration with him was crucial for completing my doctorate and would have been a lot harder without.

I would also like to thank my second supervisor Peter Jeavons for his help and support especially at the start of my DPhil and my colleages from the Oxford Constraints group Stanislav Zivný, András Salamon, Evgenij Thorstensen and Justyna Petke for interesting discussions and useful feedback. Many thanks goes also to my former supervisor Georg Friedrich for providing me with the needed support in making the move to Oxford.

Finally, I would like to thank my family and friends. Especially my mum has been a constant source of support throughout my whole life and this thesis would certainly not have existed without her. I am also very grateful to my good friends both in Oxford and beyond who have provided a welcome relief from work and have helped in a lot of ways to keep me going. You know who you are.

Statement of Originality

I hereby certify that I have written this thesis entirely by myself. Parts of this thesis have appeared in the following publications which have been subject to peer review:

- Aschinger, M., Drescher, C., Gottlob, G., and Vollmer, H. (2014). LoCo – A Logic for Configuration Problems. In *ACM Transactions on Computational Logic*. Accepted for publication.
- Aschinger, M., Drescher, C., and Vollmer, H. (2012). LoCo – A Logic for Configuration Problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 7378. IOS Press.

I would like to express here my thanks to my coauthors: Conrad Drescher, Georg Gottlob and Heribert Vollmer. Some of the theoretical results on the complexity of the presented formalism were obtained in collaboration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Technical Product Configuration

This work deals with the problem of technical product configuration. Flexibility and efficiency in the customization of products and services - rather than series production - has become a key factor of competitiveness in the post-industrial economy. To support customization activities by lowering product development and production costs, automated software configuration systems are increasingly used in enterprises. Solving such configuration problems is one of the major success stories of applied AI research: In industrial environments they support the configuration of complex products and, compared to manual processes, help to reduce error rates and increase throughput [Sabin and Weigel, 1998]. According to industry analysts a configurator that is able to support the whole configuration process of a product life cycle potentially reduces the costs by as much as 60% [Fleischanderl et al., 1998].

Configuration is used both in the B2C and in the B2B business models. A successful example of B2C product configuration is the case of *Dell Inc.*, where desktop and laptop computers are produced for end customers who have specified their individual wishes through an interactive configuration process. Other examples are flight search and booking engines, where quite complex constraints obtained from the users must be solved. Yet even more complex configurator tasks arise in the B2B domain. For example, railway interlocking systems for large train stations must be configured on an individual

basis containing a large number of elements with highly complex relationships [Falkner and Schreiner, 2014]. Generally saying practical applications are nowadays ubiquitous and range from the configuring of product bundles such as tourism packages [Aschinger et al., 2010] over configurable cars [Sinz et al., 2003] to the configuration of telephone communication switching units [Fleischanderl et al., 1998] — for surveys see e.g. [Sabin and Weigel, 1998] or the more recent [Junker, 2006]. By now it has become apparent that there are also manifold connections to the domain of software configuration, see e.g. [Hubaux et al., 2012].

The early work on using rule based configurators for customizing computers [McDermott, 1982] is generally seen as the field's starting point. Since then manifold general purpose AI techniques such as constraint satisfaction problem (CSP) and Boolean satisfiability (SAT) solving, heuristic search, and description logics (DLs) have successfully been applied to configuration.

Configuration research has also seen a vast number of different formalizations and reasoning methods being put forward. Naturally, these differ considerably with regard to expressive power and the reasoning tasks supported. Of course, the fundamental problem of configuration finding is supported by virtually all approaches, although some are tailored more towards autonomous reasoning whereas others focus on interactive reasoning [Schneeweiss and Hofstedt, 2011]. But there are other noteworthy reasoning tasks that have been studied and implemented in industrial solutions: Explanation deals with the problem of explaining to the user why a certain option is no longer available — see e.g. [Junker, 2004]. Optimization addresses the task of finding not just any but the best configuration according to some criterion (or even several thereof). Finally, reconfiguration deals with the problem of how to modify an existing configuration so as to meet some additional constraints or a new objective [Friedrich et al., 2011]. In this work we mainly deal with the problem of configuration finding and only occasionally hint at the other reasoning tasks.

1.2 The case for LoCo

In the classical definition of a configuration problem the number of components to be used is fixed [Mittal and Frayman, 1989]. In many practical configuration problems, however, the number of components needed for a solution is often unknown beforehand; for example, for some components this number depends on the choices made for other components or on changing customer requirements. One can think of this as creating new components on-the-fly throughout the solving process. Existing knowledge representation (KR) formalisms expressive enough to deal with this dynamic aspect of configuration require that explicit bounds on all generated components are given. In addition to that the usage of these formalisms usually requires the availability of a skilled person with extensive knowledge about the underlying solving algorithms. To date there is still a lack of configuration tools being able to cope with these demands and which allow a high-level modelling of a configuration problem without the usually needed deep technical background.

In this work we present LoCo, a logic that has been carefully tailored to deal with the aforementioned challenges and to meet the demands of technical product configuration. LoCo serves as a high-level representation language which allows the modelling of general configuration problems in an intuitive and declarative way without the need of having knowledge about underlying solving algorithms. We identify configurations with models of the logic; hence, configuration finding becomes model finding. LoCo supports the notions of component ports and connections and allows us to describe arbitrary component topologies; it comes with a rich language for describing binary and one-to-many connections as well as constraints that must hold for connected components. Most importantly, it relaxes the requirement of placing explicit bounds on the number of components. Instead it implicitly bounds the number of components needed through the axioms and a given set of explicitly bounded components whenever possible. We employ existential counting quantifiers to indicate the number of possible connections from one component type to another; from these we derive the finite bounds. If finite bounds could not be inferred, we can derive a smallest fix for the problem: a set of components that — if bounded by hand — suffices to make the problem finite. As configurations are then

guaranteed to be finite there are no fundamental obstacles to either fully automated or interactive reasoning support.
The standard use case of LoCo looks as follows:

- The user specifies the problem in LoCo; cf. Section 3.
- It is then decided whether the specified problem is finite (admits only finite models), and, if not, possible fixes are suggested.
- After that bounds on the number of components are computed; cf. Section 3.5.
- Finally, the specification is translated to executable code. LoCo allows translations into different target languages. In Section 5.2 we touch upon a translation into answer set programming.

The reasoning problems that we study are hence the following: (1) Decide whether the problem is finite. (2) Compute a smallest set of components sufficient to make the problem finite (if necessary). (3) Find a model/configuration.

In principle the LoCo formalism supports interactive configuration scenarios by being able to check the finiteness of a (partial) configuration. As LoCo allows the user to specify partial configurations to be used as starting point the task of interactive configuration can be reduced to a series of configuration finding problems. The system guides the user step by step through the configuration process and gives assistance in the search for valid variable assignments. The user then is able to specifically select component instances, attribute values and component connections with the system giving feedback if the resulting model remains to be finite. In case of a resulting infinite model the system recommends a smallest fix, i.e. the smallest set of components to make the model finite. These features will be discussed in section 3.5.

The feature of specifying partial configurations as part of the input can also be used to support a limited form of reconfiguration; this works as long as there is no conflict/ inconsistency between the (partial) legacy instance and the new constraint/objective. LoCo currently doesn't support full reconfiguration scenarios in the sense of [Friedrich et al., 2011] where parts of existing legacy configurations are reused or modified in order

to adapt to changes of requirements specifications. Dealing with the case of an actual inconsistency requires further research and is a promising subject of future work as is the reasoning task of explaining configuration results to end-users.

LoCo has originally been introduced in [Aschinger et al., 2011b] and in [Aschinger et al., 2012] we have elaborated upon the above mentioned reasoning tasks. The journal article [Aschinger et al., 2014] summarizes the previous publications, extends the language by additional axiom types and provides complexity results on the LoCo satisfiability and the tightness of component bounds. This article also contains the bulk of chapter 3 which represents the core of this thesis.

1.3 The House Problem — Running Example

As a running example we use a simplified version of the House Problem that we received from our industrial partner Siemens [Bettex et al., 2009]. The House Problem reflects constraints and properties that can be found in a wide range of problems involving the design and assembly of complex systems and software processes. It is basically a toy problem derived from a real-world configuration problem in close analogy to the rack configuration problem; a layered version of bin packing with side constraints [Kiziltan and Hnich, 2001].

The original rack configuration problem consists of plugging a set of electronic cards into racks with electronic connectors, outlined in [Hentenryck, 1999]. The modified and extended scenario of our industrial partner deals with assembling entire digital switching systems from racks and modules in the telecommunications sector. Typical problems consist of approximately 200 racks, 1000 frames, 30000 modules and 10000 cables with top-end configuration solutions comprising around 43000 components with 215000 attributes and 112000 ports [Fleischanderl et al., 1998].

The task of the House Problem is to put things of various types and sizes into cabinets which have to be stored in rooms of the house. For brevity and the sake of illustration, we cover only certain parts of the problem that we think are particularly helpful in understanding the underlying formalism. A cabinet has two shelves, each providing a certain storage space for either things of type A or B. Constraints on component attributes

determine where a thing or a cabinet can be stored: Big things can only be stored in big cabinets whereas some cabinet need to be located at a certain position in a room; in the case of two small cabinets one can possibly be placed on top of the other in the same position. Every thing is owned by a person and things of different persons cannot be placed together in the same room. The goal is to find a minimal number of cabinets, counting twice all big cabinets.

Figure 1.1 depicts an example scenario for the House Problem, including the basic (binary) connections between components together with cardinalities restricting the number of potential connections. Connections marked as *(Input)* are pre-defined, i.e. they are already specified from the start in a given problem instance and are therefore part of the input.

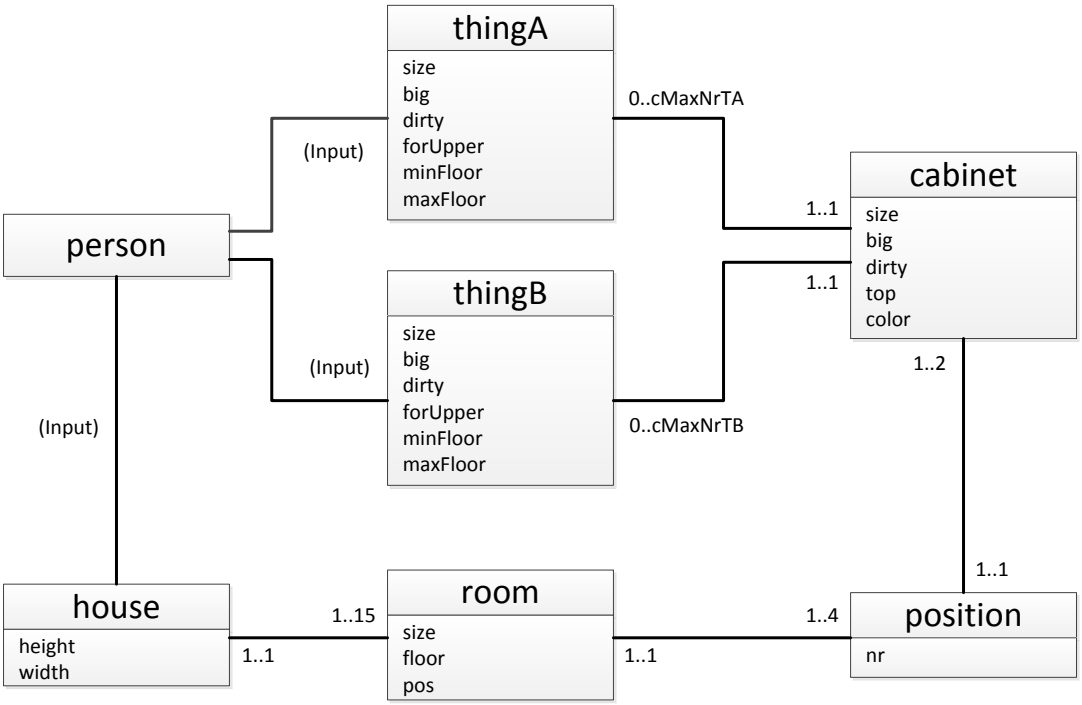


Figure 1.1: House problem scenario

1.4 Thesis Outline

The outline of the thesis is as follows: In chapter 2 we provide a more detailed definition of configuration systems together with a summary of the main approaches related to our own formalism in terms of existing problem formalizations and reasoning methods. Chapter 3 deals with the core of this thesis: We first introduce the LoCo formalism and show its use for specifying configuration problems. We furthermore discuss how to enforce that configurations contain only finitely many components and show some complexity results. In order to put the LoCo axiomatization into practice we created the LoCo input language. The basic structure of this text-based language will be presented in chapter 4. Chapter 5 gives an overview of a prototypical implementation and insights on the transformation to both ASP and MiniZinc as target languages. We then evaluate LoCo encodings on a set of benchmark problem instances that we received from our industrial partners in chapter 6. Chapter 7 finally summarises the results obtained in this thesis and discusses directions for future research.

Chapter 2

Configuration Systems

2.1 Definition

Configuration systems are one of the most successful applications of AI-techniques. In industrial environments they support the configuration of complex products and, compared to manual processes, help to reduce error rates and increase throughput [Sabin and Weigel, 1998]. Over the years there have been many diverse approaches for defining a configuration problem but none of them seems to be commonly accepted. [Stumptner, 1997, Junker, 2006] provide some overviews on this. Configuration is often viewed as related to a design problem or even as special type of it. Chandrasekaran gives a basic definition of what is typically meant by a design problem [Chandrasekaran, 1990]:

Definition 1. *A design problem is specified by*

- *a set of functions on the behavior or properties of the artefact (either stated by the design consumer or implicitly defined by the domain),*
- *a set of constraints on the properties of the artefact, the process of making the artefact or the design process itself;*
- *a repertoire of components and a vocabulary of relations between them.*

Functions are high-level constraints that describe the primary reason why the artefact is desired. An effective process of a design is to generate a candidate design based

on functions and then modify it to meet the constraints. The solution to the design problem consists of a complete specification of a set of components and their relations that together describe an artefact that delivers the functions and satisfies the constraints [Chandrasekaran, 1990].

Sabin and Weigel emphasize the connection between design and configuration, shown in definition 2 [Sabin and Weigel, 1998]. Configuration hence is a special type of design activity, with the key feature that the artifact being configured is assembled from a set of pre-defined components. This is also the main difference to a design problem, where new components can be generated during the composition of the desired artifact. This rather abstract definition is not commonly accepted though. D.C. Brown for example understands a design process mainly as a refinement of abstract components through the specification of values of their defined attributes. From this viewpoint configuration is not a special form of design but an essential part of the design process itself [Brown, 1998].

Definition 2. *Configuration can be defined as a special case of design activity where the artifact being configured is assembled from a set of well-defined component types which can be composed conforming to a set of constraints.*

Brown and Chandrasekaran divide design problems based on their complexity in 3 classes [Brown and Chandrasekaran, 1989]: The design of a new artefact completely from scratch is represented by class 1. Class 2 covers artefacts that can be decomposed in subcomponents where the structure of at least some of these components remains unknown at the beginning of the process. In case all the subcomponents are predefined then the problem belongs to class 3. A design problem of class 3 is what the authors then define as a configuration problem.

The following definition by Mittal and Frayman [Mittal and Frayman, 1989] is the most cited and describes what is typically meant by a configuration problem:

Definition 3 (Configuration Problem)**. Given:** *A fixed, predefined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints, some description of the desired configuration*

and some criteria for making optimal selections.
Build: *One or more configurations that satisfy all the requirements, where a configuration is a set of components and a description of the connections between the components in the set, or, detect inconsistencies in the requirements.*

Please note that this classical definition of a configuration problem still leaves considerable wiggle room as for how exactly the problem is to be formalized. It is also worth pointing out that in this definition components are not only pre-defined in terms of their structure but that also their number available to build the final configuration is fixed; that is, it cannot be changed during runtime. In practice, however, the number of components needed is often not easily stated beforehand. Mittal and Frayman emphasize in particular 3 important aspects of a configuration:

- New component types cannot be designed throughout the configuration process.
- Each component is pre-defined in terms of its compatibility to other components and dynamic modifications of these relations are not possible.
- A solution not only specifies the components contained in a configuration, but also how they are connected to each other.

Brown argues that not every configuration necessarily needs to consist of components that are connected physically to each other. Furthermore, it remains unclear on which abstraction level the components have to be pre-defined and if all or only a subset of the components have to be used in the configuration process [Brown, 1998]. Summing up it becomes clear at this point that there are many facets of configuration and there exist different opinions of what a configuration exactly is. The definition of Mittal and Frayman though stood the test of time and can still be seen as the most appropriate so far.

Regardless of the chosen representation approach, every configurator requires access to specific knowledge that states which combinations of components are allowed and which restrictions need to be observed. For instance, a car configurator must compute a valid vehicle variant satisfying the user requirements and all applicable commercial and technical restrictions derived from the marketing and engineering policies of the

manufacturer. The basic idea for representing configuration knowledge is to have some sort of *component catalogue* representing the space of all possible combinations of product components [Soininen and Niemelä, 1998]. It contains a description of the components along with their related attributes. The component catalogue (aka configuration space) is then restricted to the solution space by constraints which represent restrictions on how components can be combined. These rules can be either fixed domain constraints or specific user requirements.

Given a component catalogue and a set of user requirements the task of a *configuration process* is to find a configuration that satisfies the requirements. The process of configuration finding is about choosing a set of components from a component catalogue and connecting them in a way such that some predefined constraints are satisfied. A *configuration* then is a set of selected component instances of the component catalogue. Wielinga and Schreiber [Wielinga and Schreiber, 1997] divide configurations into three classes:

- A configuration satisfying all constraints of the configuration model is called *valid*;
- A valid configuration which also satisfies all user requirements is called *suitable*;
- Finally an *optimal* configuration satisfies some optimality criteria in addition to being suitable.

2.2 Constraint-based approaches

Over the years several different approaches for configuration have been investigated, e.g. expert systems, rule-based systems, non-monotonic reasoning, case-based reasoning, description logics and constraint processing. A recent survey is given by Junker in [Junker, 2006].

Configurators that utilize the constraint satisfaction problem (CSP) paradigm are within the family of model-based approaches that strictly separate domain knowledge from problem solving knowledge [Mailharro, 1998]. This increases the quality of knowledge representation since changes in the knowledge base don't have side-effects on the underlying solving engine and vice versa. Constraint satisfaction problems are currently

the most widely used approach for the formalisation of configuration problems.

2.2.1 Standard CSP

In its simplest form the problem is formalized as a standard CSP and existing mature solver technology is exploited. Contemporary research in this direction includes e.g. the compilation of the CSP for fast interactive reasoning [Amilhastre et al., 2002, Andersen et al., 2010] or the use of global constraints for greater deductive power [Karatas et al., 2010]. Note that these approaches do not come with explicit support for ports or connections between components; also, the number of individual components available to build the final configuration has to be fixed prior to solving.
A common definition for classical CSPs is shown in the following definition [Tsang, 1993]:

Definition 4. *A* CSP $\mathcal{P}$ *is defined as a triplet* $\mathcal{P} =< V, D, C >$*, where:*

- $\mathcal{V} = \{V_1, ..., V_n\}$ *is the set of variables involved in* $\mathcal{P}$*,*
- $\mathcal{D} = \{D_1, ..., D_n\}$ *is the set of domains associated to variables,* V_i *has domain* D_i *and*
- $\mathcal{C} = \{C_1, ..., C_m\}$ *is the set of constraints which must be satisfied for any solution of* $\mathcal{P}$*. A constraint* C_i *involving a set of variables* $\mathcal{W}_i = \{V_{i_1}, ..., V_{i_j}\} \subseteq \mathcal{V}$ *is defined by a tuple (scope, def). The scope of a constraint, scope*(C_i)*, is the set of variables* $\mathcal{W}_i$ *involved in the constraint.* $def(C_i)$ *is a set of value tuples for the set of variables* $\mathcal{W}_i = \{V_{i_1}, ..., V_{i_j}\}$*,* $def(C_i) \subseteq D_{i_1} \times ... \times D_{i_j}$*.* $def(C_i)$ *is called the definition of the constraint and denotes the value tuples that satisfy* C_i*.*

Given that the aim of configuration is to define proper sets of variables (e.g., the different components of a product or service) and constrain them to assume certain values (e.g., restricting the way they can be combined with each other), it comes as no surprise that Constraint Satisfaction techniques play a crucial role in this context. However, the standard CSP formulation does not feature variables or sub-CSPs that are conditionally activated depending upon the values assigned to other variables. This is because CSP instances are intrinsically static, since *(i)* they are defined over a *fixed* set

of variables and a *fixed* set of constraints on these variables, and *(ii)* they assume that all the values in the domains are known beforehand. Therefore, some kind of conditional variable activation is required.

2.2.2 Csp extensions

Hence, in the area of constraint-based configuration, a number of extensions of the traditional Csp paradigm have been developed to cope with the dynamic aspects of configuration problems: the problem is solved by incrementally adding components and selecting values for their attributes until the overall requirements are met. In such an approach it is easier to express that the exact number of components to be included in the final configuration is not fixed but depends on the choices made for the previous components.

Dynamic CSP

In Dynamic CSPs (DCSP) [Mittal and Falkenhainer, 1990], also known as Conditional Csps in the literature, activation constraints ensure that only a relevant subset of the variables and constraints is used for generating a solution.

Definition 5. *A DSCP has the form* $< V, D, V_I, C_C, C_A >$*, where* $V = \{v_1, \ldots, v_n\}$ *represents the set of variables,* $D = \{D_1, \ldots, D_n\}$ *the set of variable domains and* $V_I \subseteq V$ *the set of initially active variables. The set of constraints is divided in compatibility constraints* C_C *and activity constraints* C_A *[Mittal and Falkenhainer, 1990].*

In contrast to a standard Csp only the active variables V_I need to have assigned values. V_I contains a certain initial set of variables at the beginning of the solving process which then dynamically changes throughout the process due to state changes. Activity constraints are changing the state of variables and define the conditions for when a variable gets activated or deactivated. When a variable gets activated it needs to receive a valid assignment and is from this point on a part of the solution set. Inactive variables are of no relevance [Soininen and Gelle, 1999]. Compatibility constraints have the same function and structure as constraints of a standard Csp and are active if all involved variables are also active. Inactive compatibility constraints won't be considered in the solution finding process.

There are different approaches for solving a DCSP [Mouhoub and Sukpan, 2007]: A DCSP can be converted to a CSP by reformulating activity constraints in compatibility constraints. Another way is to generate all possible CSPs and solve them with standard algorithms. The direct way of solving a DCSP is done in form of an activation-propagation cycle [Gelle and Weigel, 1996]: Based on the active variables all activity constraints will be checked which changes the problem space. The following propagation step checks the fulfillment of all compatibility constraints. In case of an inconsistency the process backtracks to the last valid problem space. The algorithm terminates if the current variable assignments are consistent and there are no new variables that have to be activated.

In summary the main benefits of the DCSP approach are on the knowledge representation level. While it allows to map a problem structure in a more natural way, the dynamic aspects though are limited in that all possible occurring variables still need to be stated a priori [Stumptner, 1997]. This means that it remains very difficult to model a problem scenario where a potentially very large and a priori not exactly definable set of components is necessary for finding a solution. The initial consideration of the maximally needed number of components for a given problem is often either impossible or would result in a very large search space [Mailharro, 1998].

Composite CSP

Composite CSP (CCSP) is another formalism which was informally introduced by Sabin and Freuder [Sabin and Freuder, 1996]. It allows to model CSPs where variables can have subproblems (sub-CSPs) as values. If a subproblem T is assigned to a variable v then any constraint containing v is removed and the constraints and variables in T are added to the CSP. This approach provides an elegant way of decomposing a problem into a set of subproblems. According to [Mailharro, 1998] the biggest issue of a CCSP is consistency maintenance: The central question is how the knowledge about subproblems can be used to decide which of them are inconsistent in regards to the remaining variable assignments and have to be removed by a propagation algorithm.

However, the major issue remains that in both the DCSP and CCSP formalisms the numbers of possibly activated variables and constraints have to be defined in advance.

For this reason it comes as no surprise that these formalisms in fact have the same expressive power as classic CSPs. There exist polynomial-time many-one reductions between composite CSPs and dynamic CSPs as well as between dynamic CSPs and classic CSPs. Further details on the interreducibility of these formalisms including proofs and a discussion can be found in [Thorstensen, 2010].

Generative CSP

Generative CSPs (GCSP) [Stumptner et al., 1998, Stumptner and Haselböck, 1993] overcome the main limitation of Composite and Dynamic CSPs and allow the dynamic generation of components on demand during the search process. It is the only one of the dynamic CSP extensions that comes with explicit support for ports and component connections. GCSPs are based on the standard component-port model (see definition 3) and contain object-oriented concepts such as generic constraints on component types that have to hold for all derived component instances. Special resource constraints can be defined on an a priori unknown number of variables which are able to create new ones on demand. As an example consider a PC configuration where a certain amount of memory is needed but the number and type of needed memory chips is unknown at the beginning.

The reasoning starts from certain key components and then required auxiliary components and associated connections are incrementally added. New attributes are generated by activity constraints and new components are created by resource constraints. GCSPs don't require explicit bounds on the number of components and because of this the formalism allows infinite configurations to be constructed. This is the main drawback of the GCSP approach and also became one of our starting motivations for developing the LoCo formalism.

An example of a configuration system implementing the GCSP approach is COCOS [Stumptner et al., 1994]. This system focuses on solving large configurations with a high number of components and was successfully used in problem scenarios with around 4000 components in the solution set. The industrial rack configuration problem mentioned in section 1.3 for example was modelled with the LAVA configurator which is an enhanced and further developed version of COCOS.

2.3 Logic-based approaches

Complementary to the CSP formalism and its variations there has also been substantial research to capture configuration with logic-based formalisms. Here, the conditional inclusion of components into configurations is commonly modelled using implication and/or a form of existential quantification, a combination that easily leads to infinite models / configurations. We recall these in some detail, as some of them serve as kind of starting points for our own configuration logic.

2.3.1 $\exists FO_{\rightarrow,\wedge,+}$

Standard CSPs can be equivalently formulated in logic and correspond to the fragment $\exists FO_{\wedge,+}$ of $FO_{\wedge,+}$ of FO consisting of formulae built using only existential quantification and conjunction, i.e. the fragment of first-order logic of formulae with arbitrary quantifications and conjunctions, but without negation or disjunction [Kolaitis and Vardi, 1998]:

Definition 6. *The logic-based characterization of a CSP instance is defined as a pair $(\phi, \mathcal{D})$, where $\mathcal{D}$ is the constraint database, i.e., the set of all the constraint relations r_i, for each constraint $C_i = (S_i, r_i)$, and ϕ is a $\exists FO_{\wedge,+}$ sentence, i.e., an existentially quantified first-order formula with no negations or disjunctions, over the relational vocabulary consisting of the atoms $r_i(S_i)$. Solving the CSP corresponds to deciding whether $\mathcal{D} \models \phi$.*

The standard CSP formulation $\exists FO_{\wedge,+}$ of $FO_{\wedge,+}$, as already mentioned, is not appropriate for configuration problems in a knowledge representation (KR) sense: it does not feature variables that are conditionally activated depending upon the values of the other components in the solution.

In the work by Gottlob et al. [Gottlob et al., 2007] logical implication has been added to this formalism in order to express the conditional inclusion of components into configurations. The resulting language is the fragment $\exists FO_{\rightarrow,\wedge,+}$ of FO, containing formulae of the form $\exists \vec{X} \ \phi(\vec{X})$. It proposes to model the conditional inclusion of components by evaluating a strictly positive, existentially quantified first order sentence

formed by using conjunction and a restricted form of implication over an extensional finite constraint database.

Unlike for $\exists FO_{\wedge,+}$, formulas of the more expressive $\exists FO_{\rightarrow,\wedge,+}$ logic may be satisfied by partial assignments. For instance the formula $\exists X((r(X) \wedge p(X)) \implies \exists Y q(X,Y))$ is satisfied by any partial assignment mapping X to a value contained in the relation associated with r but not contained in the relation associated with p. The variable Y then is said to be *non-active* in this assignment.
This formalism allows to model constraints for example in the following way:

Example 1. *In a potential rack configuration scenario, the constraint stating that a compatible auxiliary power supplier module needs to be put in the same frame where the module of type* t1 *is plugged may be expressed by the following* $\exists FO_{\rightarrow,\wedge,+}$ *formula:*

$$\begin{aligned}
&\exists F, M \ frame(F) \wedge module(M) \wedge \\
&\quad type(M, t1) \wedge pluggedInto(M, F) \implies \\
&\qquad \exists P \ module(P) \wedge type(P, power_supplier) \wedge \\
&\qquad\quad compatible(P, M) \wedge pluggedInto(P, F)
\end{aligned}$$

A drawback of $\exists FO_{\rightarrow,\wedge,+}$ is that explicit bounds on the number of components needed have to be given (variables have a fixed finite domain) and that all constraints must be coded in extension in the constraint database. Moreover, the number of all potentially activated variables (components) must be known beforehand, too. From this it follows that it would have to be analyzed which further extensions are required in order to obtain a more expressive and versatile logical language. The $\exists FO_{\rightarrow,\wedge,+}$ logical fragment turns out to be a notational variant of dynamic and composite (and hence also standard) Csp [Thorstensen, 2010]: like these it does not feature support for ports or connections either. However, this work has pioneered the identification of families of such conditional configuration (optimization) problems that admit tractable reasoning / efficient processing. The language is – even without further extensions – sufficiently strong for modelling a wide range of configuration constraints and serves as one of the main starting points for our own formalism.

Finally, in [Friedrich and Stumptner, 1999] a logic-based formulation of GCSPs has

been given; like in the original GCSP this formulation does not require stating explicit bounds on the component numbers but admits infinite configuration models.

2.3.2 Description Logics

There are also two prominent formalisms based on Description Logics (DLs): the works by McGuinness et al. [McGuinness and Wright, 1998] and Klein et al. [Buchheit et al., 1995]. These are also the other two starting points of our formalism. In both works valid configurations are described using DL axioms. DLs are fragments of FO based on unary and binary predicates, so-called concepts and roles. Concepts are used for describing components and attributes; roles are used to describe the relations between components and also between components and attributes.

Description Logics have their origin in semantic nets and frames. In contrast to their predecessors they support in addition to the taxonomic structure also expressive logic-based semantics [Nardi and Brachman, 2003]. Based on the clear semantics and its simple logical operations DLs became a very popular research area both in a theoretical and practical way. For example AT&T became a pioneer for developing several DL-based configuration applications in the early 90s by using the knowledge-representation tool CLASSIC [Borgida et al., 1989, Wright et al., 1993].

Description Logics support a configuration process in both the knowledge acquisition and in the problem solving phase [Sabin and Weigel, 1998]. In the knowledge acquisition phase concepts can be organised automatically in an explicit taxonomy via classification. With regard to the solving phase there are two variants of support [Wright et al., 1995]: The first option is a run-time support for other configuration systems by simply providing an efficient concept taxonomy. The second option covers the whole configuration process. The two mentioned formalisms related to our own work are examples of the latter. [Buchheit et al., 1995] reduce the task of finding a valid configuration to the problem of constructing a *finite* model of a set of logical axioms. [McGuinness and Wright, 1998] propose an interactive approach where (1) the knowledge engineer adds atomic propositions to the axioms and (2) the inference engine computes the consequences until (3) eventually a finite model is obtained.

The main advantages of using a DL-based approach are the clear and simple logical operations, the automatic organisation of concepts in an explicit taxonomy and the consistency maintenance [McGuinness and Wright, 1998]. DLs automatically deduce all logical conclusions from a newly added information and subsequently detect all elements leading to inconsistencies. The potential consequences of a partial configuration selection on the remaining elements of the model are then easy to determine. Because of this DLs are well-suited for interactive applications with an iterative refinement process through user interactions.

Next to these advantages there are also some significant shortcomings of using DLs. While on the one hand these approaches support the representation of component connections, on the other hand the so-called tree model property [Baader et al., 2003] of Description Logics is at odds with modelling configurations where the connections form non-tree structures. Also, the absence of predicates of arity greater than two can make domain encodings unnecessarily complex. In general, the models of a DL axiomatization need not be finite; hence no explicit bound on the number of components has to be given. Moreover, a potential problem is the trade-off between the efficiency of reasoning and the expressivity of the knowledge representation [McGuinness and Wright, 1998]: the integration of additional inference mechanisms and operators increases the complexity of reasoning and hence the runtime. On the reverse side a reduction of expressivity leads to a better runtime but then the representation of the problem specification becomes often very cumbersome or even inadequate. This led to the development of hybrid approaches with an integration of rule- or constraint-based approaches for dealing with complex compatiblity or numerical constraints [McGuinness, 2003]. Hybrid approaches were for example successfully applied in the PLAKON [Cunis et al., 1989] and in the previously mentioned CLASSICS project.

2.4 Unified Modeling Language (UML) approaches

Finally there are works that are using the Unified Modeling Language (UML) for specifying configuration problems [Falkner et al., 2010, Feinerer, 2013]. These two works in particular are similar in spirit to LoCo in that UML's "multiplicities" allow us to specify how many components of some type can be connected to components of some other type. From these multiplicities they also derive linear inequalities from the problem specification in order to constrain the number of components used in a configuration, an idea which has been pioneered in the context of entity relationship diagrams [Lenzerini and Nobili, 1990]. LoCo follows the same idea. In contrast to LoCo, however, these UML approaches only derive lower bounds on the number of available components and hence do not rule out arbitrarily large configurations. Such an upper bound is vital, however, if we want to defer reasoning e.g. to SAT or constraint solvers instead of the integer linear solvers used in [Falkner et al., 2010, Feinerer, 2013].[1] Another weakness of these approaches is their very limited support for component attributes and consequently for expressing constraints on these.

[1] Considering configurations/databases of arbitrary size has further theoretical consequences: For example, Lenzerini and Nobili can reduce their notion of strong satisfiability (a legal database instance with at least some non-empty relations exists) to the existence of a fully populated database (no empty relations allowed). In LoCo's approach this would not work as we cannot add arbitrarily many components.

Chapter 3

LoCo– A Logic for Configuration Problems

Remark: Major parts of this chapter were previously published in [Aschinger et al., 2014, Aschinger et al., 2012, Aschinger et al., 2011b] and have been reproduced with permission.

We now introduce the core of LoCo, a new logic-based framework for modelling practical configuration problems. The basic idea is to describe a configuration problem (the problem domain) by a set of logical sentences. The task of finding a configuration is then reduced to the problem of finding a model for the logical sentences – this is the same approach as the one taken by Klein et al. [Buchheit et al., 1995]. From Gottlob et al. [Gottlob et al., 2007] we take the idea to express the conditional existence of components in configurations via implication and existential quantifiers. However, we use counting quantifiers for this, and these are already present in the work by McGuinness et al. (albeit used for a different purpose) [McGuinness and Wright, 1998]. The main idea of LoCo is that via these counting quantifiers we can enforce that each model of the configuration problem contains finitely many components only.

3.1 Introducing LoCo

Formally, LoCo is a fragment of classical first order logic with equality interpreted as identity. We also use existential counting quantifiers and a variant of sorts for terms, but both these extensions reduce to basic first order logic.

Components: Each of the different component types is modelled as an n-ary predicate $\mathit{Component}(id, \vec{x})$. Here id is the component's identifier, and $\vec{x}$ a vector of further component attributes.

Sorted Attributes: The component attributes belong to different sorts — e.g. numbers, strings, etc. Using sorted variables and terms simplifies notation. In particular, for each component type we introduce one sort ID for the identifiers. We stipulate that the finitely many different attribute sorts are all mutually disjoint.

We now show how our sorts can be accommodated in classical first order logic — this is very similar to the reduction of classical many-sorted logic to pure first order logic (cf. e.g. [Enderton, 1972]). We first introduce unary predicates for each sort (e.g. ID for sort ID) and add domain partitioning axioms:

$$(\forall x) \bigvee_{S \in \mathcal{SORTS}} S(x),$$
$$(\forall x) \bigwedge_{S_i, S_j \in \mathcal{SORTS}, i \neq j} \neg(S_i(x) \wedge S_j(x)).$$

Then, in a sorted formula, we replace each subformula $(\forall\, id)\, \phi(id)$, where the universal quantifier ranges over component identifiers only, by $(\forall\, x)\, \mathrm{ID}(x) \Rightarrow \phi(x)$ and likewise $(\exists\, id)\, \phi(id)$ by $(\exists\, x)\, \mathrm{ID}(x) \wedge \phi(x)$ — this is the standard reduction from many-sorted to classical FO. We postpone the discussion of how to treat sorted terms until Section 3.4.

Counting Quantifiers: For restricting the number of potential connections between components we use existential counting quantifiers $\exists_l^u$ with lower and upper bounds l

and u such that $l \leq u$, $l \geq 0$ and $u > 0$. For example, a formula $(\exists_l^u x)\, \phi(x)$ enforces that the number of different x (here x denotes a vector of variables), such that $\phi(x)$ holds, is restricted to be within the range $[l, u]$. In classical logic without counting quantifiers this can be expressed as

$$\bigvee_{l \leq n \leq u} \big[\, (\exists\, x_1, x_2, \ldots, x_n)\, [\, \phi(x_1) \wedge \phi(x_2) \wedge \ldots \wedge \phi(x_n)\,] \wedge \\ [\bigwedge_{i,j \in \{1..n\}, i \neq j} x_i \neq x_j\,] \wedge [\, (\forall\, x)\, [\, \phi(x) \Rightarrow \bigvee_{i \in \{1..n\}} x = x_i\,]\,]\, \big].$$

As usual sorted quantifiers range over a single sort only. But occasionally, by an abuse of notation, we will write e.g. $(\exists_l^u x)\, \phi(x) \vee \psi(x)$, where ϕ and ψ expect different sorts. This abbreviates a formula enforcing that the total number of objects such that ϕ or ψ is between l and u, where the disjunction is inclusive.

$$\bigvee_{l \leq n \leq u} \big[\, (\exists\, x_1, x_2, \ldots, x_n)\, [\, (\phi(x_1) \vee \psi(x_1)) \wedge (\phi(x_2) \vee \psi(x_2)) \wedge \ldots \wedge (\phi(x_n) \vee \psi(x_n))\,] \wedge \\ [\bigwedge_{i,j \in \{1..n\}, i \neq j} x_i \neq x_j\,] \wedge \\ [\, (\forall\, x)\, [\, \phi(x) \Rightarrow (\bigvee_{i \in \{1..n\}} x = x_i\,)\,]\,] \wedge \\ [\, (\forall\, x)\, [\, \psi(x) \Rightarrow (\bigvee_{i \in \{1..n\}} x = x_i\,)\,]\,]\, \big].$$

3.2 Connection Axioms

Configuration is about connecting components: For every set $\{C_1, C_2\}$ of potentially connected components we introduce one of the binary predicate symbols $C_1\_C_2$ and $C_2\_C_1$, where predicate $C_i\_C_j$ is of sort $\text{ID}_i \times \text{ID}_j$.[1] We allow connections from a component type to itself, i.e. via a binary predicate $C_i\_C_i$ of sort $\text{ID}_i \times \text{ID}_i$.

3.2.1 Binary connections

Standard binary connections Connections between two component types are axiomatized as follows:[2]

$$(\forall\, id_1, \vec{x})\ C_1(id_1, \vec{x}) \Rightarrow \qquad (3.1)$$
$$(\exists_{l_1}^{u_1}\, id_2)\ [\, C_1\_C_2(id_1, id_2) \wedge C_2(id_2, \vec{y}) \wedge \phi(id_1, id_2, \vec{x}, \vec{y}\,)\,]$$

This axiom specifies how many components of type C_2 can be connected to any given component of type C_1. The purpose of the subformula ϕ (with variables among $id_1, id_2, \vec{x}, \vec{y}$) is to express additional constraints, like e.g. an aggregate function $\sum n \leq$ *Capacity*. For these constraints we allow ϕ to be a Boolean combination of arithmetic expressions and attribute comparisons ($<, =, ...$) over a subset of all quantified variables of the axiom. Generally the type of supported subformulas is restricted to the expressiveness of the chosen target output language. For special constructs like e.g. aggregate functions SUM and COUNT we perform language-specific transformations. Section 5.2 contains a description of Answer Set Programming as the target output language for constraint elements.

Whenever possible an axiom for the reverse direction should be included, too. This is especially important for the proper computation of component bounds as we shall see later on:

[1] Note that this precludes having multiple different connection relationships between two different component types.

[2] Throughout this manuscript free variables in formulas are to be read as existentially quantified.

$$(\forall\, id_2, \vec{x})\ C_2(id_2, \vec{x}) \Rightarrow \qquad (3.2)$$
$$(\exists_{l_2}^{u_2}\, id_1)\ [\, C_1\_C_2(id_1, id_2) \wedge C_1(id_1, \vec{y}) \wedge \psi(id_1, id_2, \vec{x}, \vec{y})\,]$$

We stipulate that the upper bound of the counting quantifier is greater than zero in all connection axioms; an omitted upper bound means arbitrarily many components may be connected whereas an omitted lower bound is read as zero. The following example shows a basic binary connection from our running example. For ease of presentation the component attributes are quantified via a combined attribute vector $\vec{attr}$.

Example 2. *In the House Problem each thing of type A needs to be placed into exactly one cabinet. Moreover, things that are big can only be put in big cabinets — in configuration terms big things and small cabinets are not compatible:*

$$(\forall\, id_{TA}, \vec{attr}_{TA})\ thingA(id_{TA}, \vec{attr}_{TA}) \Rightarrow$$
$$(\exists_1^1\, id_C)\ \big[\, thingA\_Cab(id_{TA}, id_C) \wedge cab(id_C, \vec{attr}_C) \wedge$$
$$[\,(big_C = 1 \wedge big_{TA} = 1) \vee (big_{TA} = 0)\,]\,\big]$$

Unfolded binary connections For some configuration problems it is necessary to distinguish different cases in the binary connection axioms:

$$(\forall\, id_1, \vec{x})\ C_1(id_1, \vec{x}) \Rightarrow \qquad (3.3)$$
$$\bigvee_i\ \big[\, (\exists_{l_i}^{u_i}\, id_2)\ [\, C_1\_C_2(id_1, id_2) \wedge C_2(id_2, \vec{y}) \wedge \phi_i(id_1, id_2, \vec{x}, \vec{y})\,]\,\big],$$

where the intervals $[l_i, u_i]$ are non-overlapping in order to be mutually exclusive and $\phi_i(id_1, id_2, \vec{x}, \vec{y})$ may be a different formula for each case.[3] An even higher level of granularity can be reached by completely unfolding the existential counting quantifiers, i.e. defining a separate case for each possible number of occurring id_2 objects.

[3]Note that there are unique smallest, and biggest, l_i, and u_i, respectively.

Example 3. *When connecting positions and cabinets we wish to differentiate between the cases where exactly one or two cabinets are connected to a position:*

$$
\begin{aligned}
&(\forall\, id_P)\ pos(id_P) \Rightarrow \\
&\quad \big[\, (\exists^1_1\, id_C)\ [\, cab\_pos(id_C, id_P) \wedge cab(id_C, \vec{attr}_C) \wedge \\
&\qquad [\, top_C = 0\,]\,]\,\big] \ \vee \\
&\quad \big[\, (\exists^2_2\, id_C)\ [\, cab\_pos(id_C, id_P) \wedge cab(id_C, \vec{attr}_C) \ \wedge \\
&\qquad [\, big_C = 0 \wedge (\, (top[1]_C = 1 \wedge top[2]_C = 0) \vee (top[1]_C = 0 \wedge top[2]_C = 1)\,)\,]\,]\,\big]
\end{aligned}
$$

Each case has a separate constraint part ϕ. When lower bound equals upper bound in the counting quantifier (like in both cases of this example so that the exact number of components is known) we can address each component instance and the respective attributes individually, abbreviated here as e.g. top[1] and top[2]. While the order of the instances is not defined there needs to exist a permutation such that the constraint is satisfied. This means that each index used in the constraint part needs to match with one component instance, e.g. top[1] with the second cabinet and vice versa for top[2]. If we address an attribute without an index then this expression has to hold for all component instances containing the attribute; in the same way as in standard binary connections discussed above.

3.2.2 One-to-many connections

Next there are also rules for supporting one-to-many connections, i.e. connecting one component with a set of components. We start this section with a description of the most common form depicted in formula 3.4.

Standard one-to-many connections

$$
\begin{aligned}
&(\forall\, id_1, \vec{x})\ C(id_1, \vec{x}) \Rightarrow \\
&\qquad (\exists^u_l\, id_2)\ \Big[\, \bigvee_i [\, C\_C_i(id_1, id_2) \wedge C_i(id_2, \vec{y})\,]\ \wedge \phi(id_1, id_2, \vec{x}, \vec{y})\,\Big]
\end{aligned}
\tag{3.4}
$$

In this rule the quantifier $\exists_l^u$ ranges over the $i > 1$ different ID sorts. Note that the single component on the left hand side is not allowed to be part of the set.

Example 4. *In the House Problem a cabinet has a separate binary connection to each type of thing determining that the number of instances that can be stored lies between zero and a certain upper bound. To make sure that there are no empty cabinets in our model, the following one-to-many axiom states that each generated cabinet needs to have at least one thing placed in it:*

$$
\begin{aligned}
&(\forall\, id_C, \vec{attr}_C)\ \ cab(id_C, \vec{attr}_C) \Rightarrow \\
&\quad (\exists_1\, id_T)\ \big[\, [\, thingA\_cab(id_T, id_C) \wedge thingA(id_T, \vec{attr}_{TA})\,] \vee \\
&\qquad\qquad [\, thingB\_cab(id_T, id_C) \wedge thingB(id_T, \vec{attr}_{TB})\,]\,\big] \wedge \\
&\qquad [\, \#big_{TA} + \#big_{TB} \leq 4\,]
\end{aligned}
$$

Next to ensuring finiteness of the model the ϕ subformula of example 4 states that a cabinet can store at most 4 big things regardless of type. In order to do this we separately count the number of big things for type A and B. The '#' symbol here in this context represents a counting aggregate. The next example highlights how to combine binary and one-to-many connection axioms in order to model the common configuration task of resource balancing.

Example 5. *Assume that things of type A contribute a certain amount of some resource whereas things of type B consume this resource. The exact quantities will be described in the component catalogue (to be introduced below). We want to ensure that for each cabinet the amount of the resource contributed is greater or equal to that consumed. To this end, for both cabinets and things of type A and B we introduce an additional numerical attribute — for better readability we are going to ignore the other component attributes. As before, the binary connection axioms describe how many things of type A and B can be stored per cabinet, say between one and two each. The following one-to-many axiom ensures the resource-balancing for a cabinet:*

$$
\begin{aligned}
&(\forall\, id_C, \vec{attr}_C)\ \ cab(id_C, \vec{attr}_C) \Rightarrow \\
&\quad \big[\, (\exists_2^4\, id_T)\ \big[\, [\, thingA\_Cab(id_T, id_C) \wedge thingA(id_T, \vec{attr}_{TA})\,] \vee \\
&\qquad\qquad [\, thingB\_Cab(id_T, id_C) \wedge thingB(id_T, \vec{attr}_{TB})\,]\,\big] \wedge \\
&\qquad [\, \sum tRes_A \geq \sum tRes_B\,]\,\big]
\end{aligned}
$$

Exclusive-OR one-to-many connections There is also an exclusive-or variant of the one-to-many connection axiom. It looks as follows, with l, u the same in all disjuncts:

$$
(\forall\, id, \vec{x})\ C(id, \vec{x}) \Rightarrow \bigoplus_i\ \big[\, (\exists_l^u\, id_i)\, [\, C\_C_i(id, id_i) \wedge C_i(id_i, \vec{y})\,]\,\big] \tag{3.5}
$$

This allows the natural formulation of certain compatibility relations that otherwise would have to be formulated in LoCo's standard way for expressing compatibility relations: By using constraints attached to connection axioms. With the help of exclusive-or axioms we can easily prevent incompatible components of being linked together.

Example 6. *Either 1 to 5 things of type A or 1 to 3 things of type B can be put in a cabinet, but A and B things cannot be stored together.*

$$
\begin{aligned}
&(\forall\, id_C, \vec{attr}_C)\ \ cab(id_C, \vec{attr}_C) \Rightarrow \\
&\quad \big[\, (\exists_1^5\, id_{TA})\ [\, thingA\_cab(id_{TA}, id_C) \wedge thingA(id_{TA}, \vec{attr}_{TA})\,]\,\big] \oplus \\
&\quad \big[\, (\exists_1^3\, id_{TB})\ [\, thingB\_cab(id_{TB}, id_C) \wedge thingB(id_{TB}, \vec{attr}_{TB})\,]\,\big]
\end{aligned}
$$

We stipulate for every one-to-many connection that the component on the left-hand side needs to have binary connections coming in from all components appearing on the right-hand side. This condition is needed for the proper computation of component bounds.

General one-to-many connections For some configuration problems it may be necessary to address the individual connected components in a one-to-many connection instead of the whole set. To this end we introduce the following most general form of a one-to-many connection axiom:

$$(\forall\, id, \vec{x})\; C(id, \vec{x}) \Rightarrow \bigvee_i \Big[\Big[\bigwedge_j (\exists^{n_{i_j}}_{n_{i_j}}\, id_j)\, [\, C\_C_j(id, id_j) \wedge C_j(id_j, \vec{y}_j)\,] \Big] \wedge \phi_i(id, id_j, \vec{x}, \vec{y}_j) \Big] \tag{3.6}$$

The component C can be connected to a number of components C_j — but C cannot be among the C_j. The rule has i cases: Each case i states for each of the components C_j the exact number n_{i_j} of connections between C and C_j. Note that we allow $n_{i_j} = 0$, but there must not be two disjuncts with identical bounds n_{i_j} for all partaking components C_j; hence all the i cases are mutually exclusive. For each case there is a separate optional ϕ subformula. In case of $n_{i_j} > 1$ component instances can be addressed in the same way as previously described for unfolded binary connections. This axiom type can express the other one-to-many connection axioms as long as no upper bounds in the counting quantifier are omitted: All the different possible cases can be enumerated.

Example 7. *This example shows a potential unfolding of the one-to-many connection between cabinet and things of type A and B into different cases. Notice that the stated ϕ subformulas have no practial meaning in terms of the House Problem since it is an artifical example and their purpose is solely to demonstrate the full range of supported language elements. For example the first case requires one instance of each type of thing with the subconstraint enforcing a connection between them. In the second case we have 2 instances of A and 3 instances of B where there has to be at least one arbitrary A instance having greater size than all B instances. The subconstraint of the third case restricts the total size of type A components via an aggregate function and doesn't take the type B components into account at all; the only restriction is that there have to be*

exactly 4 arbitrary type B instances.

$$
\begin{aligned}
&(\forall\, id_C, \vec{attr}_C)\ \ cab(id_C, \vec{attr}_C) \Rightarrow \\
&\quad \big[\, (\exists_1^1\, id_{TA})\ [\, thingA\_cab(id_{TA}, id_C) \wedge thingA(id_{TA}, \vec{attr}_{TA})\,] \wedge \\
&\quad\ \ (\exists_1^1\, id_{TB})\ [\, thingB\_cab(id_{TB}, id_C) \wedge thingB(id_{TB}, \vec{attr}_{TB})\,] \wedge \\
&\quad\ \ [\, thingA\_thingB(id_{TA}, id_{TB})\,]\,\big] \vee \\
&\quad \vdots \\
&\quad \big[\, (\exists_2^2\, id_{TA})\ [\, thingA\_cab(id_{TA}, id_C) \wedge thingA(id_{TA}, \vec{attr}_{TA})\,] \wedge \\
&\quad\ \ (\exists_3^3\, id_{TB})\ [\, thingB\_cab(id_{TB}, id_C) \wedge thingB(id_{TB}, \vec{attr}_{TB})\,] \wedge \\
&\quad\ \ [\, size[1]_{TA} > size_{TB}\,]\,\big] \vee \\
&\quad \vdots \\
&\quad \big[\, (\exists_3^3\, id_{TA})\ [\, thingA\_cab(id_{TA}, id_C) \wedge thingA(id_{TA}, \vec{attr}_{TA})\,] \wedge \\
&\quad\ \ (\exists_4^4\, id_{TB})\ [\, thingB\_cab(id_{TB}, id_C) \wedge thingB(id_{TB}, \vec{attr}_{TB})\,] \wedge \\
&\quad\ \ [\, \sum size_{TA} < 5\,]\,\big]
\end{aligned}
$$

3.3 Consistency Axioms

Next to connection axioms we also support various forms of consistency axioms in order to express so-called non-local constraints. The ϕ subformulas of connection axioms we've introduced so far are local constraints, i.e. they allow to state expressions involving the components and attributes of the particular connection axiom they belong to. The non-local constraints of consistency rules on the other hand make it possible to create expressions involving attributes of components which are not necessarily connected via connection axioms.

3.3.1 Candidate key axioms

These axioms reflect the meaning of candidate keys in relational models of databases in order to express the fact that a certain combination of attributes uniquely identifies a

component, i.e. a LoCo model cannot contain two instances of a component type with the same values for these attributes. Rules of this type are axiomatized as follows:

$$(\forall\, id_1, \vec{x}_1, id_2, \vec{x}_2)\ C(id_1, \vec{x}_1) \wedge C(id_2, \vec{x}_2) \wedge \phi(id_1, \vec{x}_1, id_2, \vec{x}_2) \Rightarrow id_1 = id_2 \tag{3.7}$$

Example 8. *There must not be two rooms on the same floor and the same position in a house, i.e. rooms located on the same floor and position must be identical.*

$$(\forall\, id_{R1}, \vec{attr}_{R1}, id_{R2}, \vec{attr}_{R2})\ room(id_{R1}, \vec{attr}_{R1}) \wedge room(id_{R2}, \vec{attr}_{R2}) \wedge [\,(floor_{R1} = floor_{R2}) \wedge (pos_{R1} = pos_{R2})\,] \Rightarrow id_{R1} = id_{R2}$$

Example 8 shows an application for the standard case of a candidate key constraint axiom. In case two rooms, identified by id_{R1} and id_{R2} respectively, have identical *floor* and *position* attributes then the *id*'s must also be identical. The ϕ constraint subformula is bound by brackets in the same way as with connection axioms. Next to the standard case outlined above we also support a slight extension of the candidate key constraint idea in that we can also take connections to other components as conditions into account.

Example 9. *In the House Problem two positions which have the same number value and are connected to the same room must be identical.*

$$\begin{aligned}(\forall\, & id_{P1}, nr_{P1}, id_{P2}, nr_{P2}, id_R) \\ & pos(id_{P1}, nr_{P1}) \wedge pos(id_{P2}, nr_{P2}) \wedge \\ & [\,(nr_{P1} = nr_{P2}) \wedge pos\_room(id_{P1}, id_R) \wedge pos\_room(id_{P2}, id_R)\,] \Rightarrow \\ & \quad id_{P1} = id_{P2}\end{aligned}$$

This form of extension is shown in Example 9. Two positions must be identical in case they are connected to the same room and also have an identical number. Checking for existing or non-existing connections in the constraint part works in the same way as checking for attribute values.

3.3.2 Connection-generating axioms

Another variant of consistency axioms are "connection-generating" axioms for expressing the fact that some connections depend on the presence of others:

$$(\forall)\ \phi(\vec{x}) \Rightarrow C_1\_C_2(id_1, id_2) \tag{3.8}$$

Here $\phi(\vec{x})$ is a Boolean combination of components, connections and arithmetic and attribute comparisons. Contrary to connection axioms this axiom is not "local": It can talk about chains of connected components of different types.

Example 10. *In the House Problem we wish to express that if a thing belonging to a person is stored in a room then the room belongs to the person. Note that things are stored in cabinets which are stored in positions belonging to rooms.*

$$\begin{aligned}(\forall)\ [\, &pers(id_{PE}) \wedge thingA(id_{TA}, \vec{attr}_{TA}) \wedge pers\_thingA(id_{PE}, id_{TA}) \wedge \\ &cab(id_C, \vec{attr}_C) \wedge thingA\_cab(id_{TA}, id_C) \wedge pos(id_{PO}) \wedge \\ &cab\_pos(id_C, id_{PO}) \wedge room(id_R) \wedge pos\_room(id_{PO}, id_R)\,] \Rightarrow \\ &room\_pers(id_R, id_{PE})\end{aligned}$$

Example 10 shows the rule for things of type A. Analogously there's a similar rule for type B. Basically axioms of this type enforce the existence of a connection provided a given set of components is connected in a certain way plus furthermore taking into account some optional side constraints. A ϕ subformula involving some attribute comparisons could for instance be added right before the main implication of example 10 in order to express

additional restrictions.

3.3.3 General First Order axioms

As a last type of consistency axiom we introduce a "general first-order" axiom (GFO) for an even higher level of expressiveness than the previous two axioms. Its general form is very simple and consists basically of two subformulas connected by an implication.

$$(\forall)\ \phi(\vec{x}) \Rightarrow \psi(\vec{x}) \tag{3.9}$$

The first subformula $\phi(\vec{x})$ represents the antecedent while the second subformula $\psi(\vec{x})$ represents the consequent of the axiom. Same as for connection-generating axioms the subformulas consist of a Boolean combination of components, connections and arithmetic and attribute comparisons. Axioms of this type are also used for an implicit representation of component catalogue knowledge. Note that both the two other forms of consistency axioms can be expressed by this most general version. The main difference is on the level of translation into a given target language as the translation of the specific axioms results in more efficient code. This will be further discussed in detail later on in chapter 5.

Example 11. *In the House Problem cabinets which are placed on positions 1 or 2 of a room need to be colored blue.*

$$\begin{aligned}(\forall\, id_C, \vec{attr}_C, id_P, \vec{attr}_P)\ \ & cab(id_C, \vec{attr}_C) \wedge pos(id_P, \vec{attr}_P) \wedge \\ & cab\_pos(id_C, id_P) \wedge [\,(nr_P == 1 \vee nr_P == 2)\,] \ \Rightarrow \\ & [\,color_C = blue\,]\end{aligned}$$

Example 11 shows a simple GFO axiom. The restriction to positions with number 1 or 2 represents the $\phi(\vec{x})$ pre-condition while the consequent stating that the cabinet needs to be colored blue represents the $\psi(\vec{x})$ post-condition. Notice that in this case we

could alternatively express the same knowledge in a ϕ constraint of the binary connection between a cabinet and a position. This is generally the case for most of the examples as LoCo allows to model a problem in several ways by using different combinations of axioms. Example 11 in particular though is a typical example for an intensional component catalogue axiom and so the actual fact is best defined in the presented form.

The next example shows a scenario where using a GFO axiom is the only suitable way of formalizing the needed knowledge in LoCo.

Example 12. *In the House Problem a room can only contain things which are allowed of being stored on the room's floor level.*

$$
\begin{aligned}
&(\forall\, id_{TB}, \vec{attr}_{TB}, id_C, \vec{attr}_C, id_P, \vec{attr}_P, id_R, \vec{attr}_R) \\
&\quad thingB(id_{TB}, \vec{attr}_{TB}) \wedge cab(id_C, \vec{attr}_C) \wedge pos(id_P, \vec{attr}_P) \wedge room(id_R, \vec{attr}_R) \wedge \\
&\quad thingB\_cab(id_{TB}, id_C) \wedge cab\_pos(id_C, id_P) \wedge pos\_room(id_P, id_R) \;\Rightarrow \\
&\qquad [\, minFloor_{TB} \leq floor_R \wedge maxFloor_{TB} \geq floor_R \,]
\end{aligned}
$$

Example 12 represents the constraint that a thing (in this case of type B) can only be stored on an admissible level. The attributes $minFloor_{TB}$ and $maxFloor_{TB}$ delimit the valid interval of allowed levels of a thing whereas the actual level is represented by attribute $floor_R$ of a room. A thing can only be stored in a given room if the room's level is in the interval between $minFloor_{TB}$ and $maxFloor_{TB}$. The GFO axiom combines the binary connections starting from a thing to a cabinet, a cabinet to a position and finally a position to a room in the antecedent and states the required attribute comparisons in the consequent. The chain of binary connections represents $\phi(\vec{x})$ while the conjunction of inequalities represents $\psi(\vec{x})$.

GFO axioms also allow the introduction of existentially quantified variables in the consequent. In contrast to all other axioms using existential counting quantifiers axioms of this type use standard existential quantifiers $\exists^{\geq 1}$ expressing the fact that there needs to exist at least 1 instance of a certain component type.

Example 13. *For every thingA component connected with a cabinet there exists a person who is the owner of this thing.*

$$(\forall\, id_{TA}, \vec{attr}_{TA}, id_C, \vec{attr}_C)\ \ thingA(id_{TA}, \vec{attr}_{TA}) \wedge cab(id_C, \vec{attr}_C) \wedge$$
$$[\, thingA\_cab(id_{TA}, id_C) \wedge dirty_{TA} = dirty_C \,] \Rightarrow$$
$$(\exists\, id_P)\ \ person(id_P) \wedge [\, person\_thing(id_P, id_{TA}) \,]$$

Discussion

Finally let us point out the following: As long as LoCo configurations are guaranteed to be finite (see below for how this is achieved) configuration finding reduces to model construction over a finite universe. Hence in principle the inclusion of arbitrary first order axioms into the axiomatization in order to express requirements that can otherwise not be stated does not lead to undecidability or infinite configurations. In fact, including infinity axioms only results in an unsatisfiable problem.

3.4 Specifying Configuration Problems

The specification of a configuration problem in our logic consists of two parts:

- domain knowledge in the form of the connection axioms, naming schemes, a component catalogue and an axiomatisation of arithmetic; and
- instance knowledge in the form of component domain axioms.

Below we will speak of *input* and *generated* components. The intuition is that only for the former we know exactly how many are used in a configuration from the beginning. We stipulate that a configuration problem always includes at least one component of the input variant.

3.4.1 Domain Knowledge

The LoCo domain knowledge consists of connection axioms, a specification of the attribute ranges and the component catalogue.

Connection Axioms Connection axioms take the form introduced above. Let us next briefly elaborate on how to model the concept of a port in LoCo.

Ports Component ports are modelled as individual components in LoCo. A normal component may have many ports (i.e. be connected to many port components); however, each port belongs to exactly one component. The connection of a component port has the same structure as a binary connection axiom:

Example 14. *Position is used as a component port of a room to place cabinets in it at a certain location.*

$$(\forall\, id_R)\ room(id_R) \Rightarrow \\ (\exists^4_4\, id_P)\ [\, room\_pos(id_R, id_P) \wedge pos(id_P)\,]$$

Attribute Ranges For all attribute sorts a naming-scheme is included. For ordinary component attributes these take the form (3.10) for sort predicate S and some first order formula $\phi(x)$:

$$(\forall\, x)\ S(x) \equiv \phi(x). \tag{3.10}$$

For component attributes of sort ID the naming-scheme has the form (3.11); i.e. components are numbered:

$$(\forall\, x)\, S(x) \Rightarrow (\exists\, n)\, x = \mathrm{SName}(n). \tag{3.11}$$

The form (3.11) allows terms not to be component identifiers even if they are a component number: We introduce a sort EXCESS without naming-scheme axiom and the names of components not used in a configuration can be discarded by assigning them to

this type. Finally, for every component type we introduce an axiom

$$(\forall\, id_i, id_j, \vec{x}, \vec{y})\ [\ C(id_i, \vec{x}) \wedge C(id_j, \vec{y}) \wedge id_i = id_j\ ] \Rightarrow \vec{x} = \vec{y} \tag{3.12}$$

expressing the fact that, in database terminology, the respective ID is a *key*. Unique name axioms for all distinct ground terms are included, too. Finally, the domain knowledge might include domain dependent axiomatizations of attribute value orderings or e.g. finite-domain arithmetic.

Component Catalogue (v1) For each component type the catalogue contains information on the instances that can actually be manufactured. In LoCo this is done with an axiom:

$$(\forall\, id, \vec{x})\ C(id, \vec{x}) \equiv \bigvee_i \vec{x} = \vec{V}_i, \tag{3.13}$$

where the $\vec{V}_i$ are vectors of ground terms. If the component has no attributes the axiom is omitted. Example 15 depicts a listing of attribute tuples for thingA:

Example 15. *A potential extensional component catalogue definition for thingA looks as follows:*

$$\begin{aligned}(\forall\, id_{TA}, \vec{attr}_{TA})\ & thingA(id_{TA}, \vec{attr}_{TA}) \Rightarrow \\ & \vec{attr}_{TA} = (\,5, \mathit{TRUE}, \mathit{FALSE}, \mathit{FALSE}, 0, 2\,) \vee \\ & \vec{attr}_{TA} = (\,3, \mathit{FALSE}, \mathit{FALSE}, \mathit{TRUE}, 1, 1\,) \vee \\ & \vec{attr}_{TA} = (\,4, \mathit{FALSE}, \mathit{TRUE}, \mathit{FALSE}, 1, 3\,) \vee \\ & \ldots\end{aligned}$$

Component Catalogue (v2) The component catalogue as outlined above and introduced in [Aschinger et al., 2012] does not conform to industrial practice in that it is

extensional: Every legal combination of attribute values per component type has to be explicitly listed. In practice, however, the component catalogue is usually specified via attribute ranges and constraints that determine the legal combinations [Junker, 2006]. This kind of component catalogue can be expressed in LoCo as an axiom

$$(\forall\, id, \vec{x})\ C(id, \vec{x}) \equiv \phi(\vec{x}), \tag{3.14}$$

where $\phi(\vec{x})$ is a quantifier-free formula on attribute comparisons. A catalogue for a component type can potentially be composed of several formulas as well as of a combination of extensionally and intensionally defined knowledge. Example 16 shows a snapshot from our running example where the explicit listing of values for attribute *forUpper* is replaced by using 2 internal component catalogue axioms. This course of action reduces the size of the component catalogue significantly which holds true especially for cases with a high number of thing instances.

Example 16. *Only things with an internal ID smaller or equal to 3 are eligible to be stored in a cabinet which is placed on a top position.*

$$(\forall\, id_{TA}, \vec{attr}_{TA})\ thingA(id_{TA}, \vec{attr}_{TA}) \iff [\, id_{TA} \leq 3 \Rightarrow forUpper_{TA} = TRUE\,]$$

$$(\forall\, id_{TA}, \vec{attr}_{TA})\ thingA(id_{TA}, \vec{attr}_{TA}) \iff [\, id_{TA} > 3 \Rightarrow forUpper_{TA} = FALSE\,]$$

Next to defining attribute values we could also set attributes in relation to each other or in relation to constants such as in example 17. The two formulas of example 16 plus the formula of example 17 could alternatively also be linked together to form one big conjunction formula containing the intensional knowledge for thingA. As mentioned we can optionally mix extensional and intensional knowledge and the values for the remaining attributes could then for example be expressed via an extensional component catalogue.

Example 17. *The minFloor value has to be always smaller or equal to the maxFloor value and the size needs to be smaller than constant maxSizeTA.*

$$(\forall\, id_{TA}, \vec{attr}_{TA})\ thingA(id_{TA}, \vec{attr}_{TA}) \iff$$
$$[\, minFloor_{TA} \leq maxFloor_{TA} \wedge size_{TA} < maxSizeTA \,]$$

Such an intensional component catalogue, however, requires a different treatment of the attribute ranges. In particular, we need to ensure that per component type there are still only finitely many different possible instances (combinations of attributes). Hence we stipulate that for an intensional component catalogue the attribute ranges are to be specified by domain closure axioms of the form

$$(\forall\, x)\ S(x) \equiv \bigvee_i x = V_i,$$

where the different possible values V_i are ground terms. It is not hard to see, however, that with this approach it is NP-complete to determine whether some attribute combination conforms to the catalogue. It was this observation that led to the original definition of an extensional component catalogue in LoCo.

3.4.2 Instance Knowledge

The subdivision of the component types into components of type *input* and of type *generated* takes place on the instance level. Note that a component being *input* does not mean we have to specify all the component's attribute values, it only means we know exactly how many instances of this component we want to use.

For components C of the input variant we make a closure assumption on the domain of the components identifiers:

$$(\forall x)\ \mathrm{ID}(x) \equiv \bigvee_{\mathrm{ID}_i \in \mathcal{ID}} x = \mathrm{ID}_i. \tag{3.15}$$

where $\mathcal{ID}$ is a finite set of identifiers ID_i and ID is the respective sort predicate. This

axiom is stronger than the naming-scheme for the component; hence, if a configuration exists, identifiers mentioned in the naming-scheme axiom but not in the domain closure axiom can only belong to the sort EXCESS.

On the instance level components to be used in the configuration can be listed, too. This can be done via simple ground literals or via formulas of the form

$$(\exists)\, C(id, \vec{x}) \quad \text{or} \quad (\forall)\, \neg C(id, \vec{x})$$

where $id, \vec{x}$ may be variables or terms. Known (non-)connections can be specified via ground literals like e.g. $\neg C_1\_C_2(\mathrm{ID}_1, \mathrm{ID}_2)$. Similar to input components we support closure axioms on connections:

$$(\forall)\ C_i\_C_j(id_i, id_j) \equiv \bigvee (id_i = \mathrm{ID}_1 \wedge id_j = \mathrm{ID}_2).$$

We now summarize the above discussion of all the different LoCo features in a more generic characterisation of our logical fragment. The following definition references all the axiom schemes that make up the logical language of LoCo:

Definition 7 (Configuration Domain Axiomatization in LoCo)**.** *A configuration domain axiomatization in* LoCo *consists of domain knowledge and instance knowledge. The domain knowledge comprises*

- *connection and consistency axioms in the forms (3.1, 3.3, 3.4 — 3.9);*
- *a specification of the attribute ranges (3.10);*
- *a specification of the component identifier naming scheme (3.11);*
- *candidate key axioms for all component* ID*'s (3.12);*
- *a component catalogue in either of the forms (3.13) or (3.14); and*
- *an axiomatization of finite domain arithmetic,*

whereas the instance knowledge is made up of

- *a designation of the* input *components together with their respective count;*
- *domain closure axioms on the component identifiers (3.15);*

- *partial configurations, consisting of components and connections with possibly existentially quantified attributes; and*
- *forbidden partial configurations, consisting of components and connections with possibly universally quantified attributes.*

3.5 Enforcing Finite Configurations

Next we discuss how to enforce that configurations contain only finitely many components. In order to transform a problem model into a target language we need to know the lower and upper bounds on the number of instances for each component of the "generated" variety. For computing the possible domain sizes of generated components, we extract Diophantine inequalities from the connection formulas. This builds up on the work by Falkner et al. and Feinerer about semantics of UML class diagrams and cardinalities applied to the configuration domain [Falkner et al., 2010, Feinerer, 2013]. They propose (1) to model configuration problems via UML and (2) to solve them via integer programming. We note that LoCo is considerably more general, though.

3.5.1 Locally Bounding Component Numbers

We start by discussing in which way the connection axioms can be used to locally bound the number of components used. We disregard the "constraint formulas" ϕ and ψ for this calculation; the bounds are based only on the lower and upper bounds on the number of connections expressed in the existential counting quantifiers.

Let us first introduce some notation: Let $\mathcal{C}$ denote the set of components of type C that can be used in a configuration and let $|\mathcal{C}|$ denote this set's cardinality.

Bounds for binary connections Assume a binary connection defined by formulas (3.1) and (3.2).

$$\begin{aligned} l_1 * |\mathcal{C}_1| \leq n \leq u_1 * |\mathcal{C}_1| \\ l_2 * |\mathcal{C}_2| \leq n \leq u_2 * |\mathcal{C}_2| \end{aligned} \tag{3.16}$$

The number of possible links n between the components is bounded as shown in 3.16. From this we can derive inequalities representing the relation between $\mathcal{C}_1$ and $\mathcal{C}_2$. For component C_2 we then have:

$$l_1 * |\mathcal{C}_1| \leq u_2 * |\mathcal{C}_2| \text{ and } l_2 * |\mathcal{C}_2| \leq u_1 * |\mathcal{C}_1| \tag{3.17}$$

The first inequality of 3.17 holds because we cannot connect the elements of $\mathcal{C}_2$ to more than u_2 elements of $\mathcal{C}_1$ each, while each element of $\mathcal{C}_1$ has to be connected to at least l_1 elements of $\mathcal{C}_2$. Hence, if we have a lower bound on the cardinality of $\mathcal{C}_1$ this implies a lower bound on the cardinality of $|\mathcal{C}_2|$. The intuition behind the upper bound is analogous. Here in particular, if we have a finite upper bound on the cardinality of $\mathcal{C}_1$ and $l_2 \neq 0$ then we can derive a finite upper bound on the cardinality of $\mathcal{C}_2$. For $\mathcal{C}_1$ as an input and $\mathcal{C}_2$ as a generated component we get

$$\text{lower bound LB} = \left\lceil \frac{l_1 * |\mathcal{C}_1|}{u_2} \right\rceil \text{ and upper bound UB } = \left\lfloor \frac{u_1 * |\mathcal{C}_1|}{l_2} \right\rfloor,$$

resulting in formula 3.18 for the bounds of $\mathcal{C}_2$. We round the lower bound up to the next integer value and analogously round down the upper bound. The outlined computation also applies to connections between two generated components, provided that component $\mathcal{C}_1$ has correctly defined bounds. In this scenario we insert the lower bound of $\mathcal{C}_1$ for computing LB and the upper bound of $\mathcal{C}_1$ for computing UB of $\mathcal{C}_2$.

$$\left\lceil \frac{l_1 * \lfloor |\mathcal{C}_1| \rfloor}{u_2} \right\rceil \leq |\mathcal{C}_2| \leq \left\lfloor \frac{u_1 * \lceil |\mathcal{C}_1| \rceil}{l_2} \right\rfloor \tag{3.18}$$

Bounds for one-to-many connections Next assume we have a basic one-to-many connection axiom (3.4) from C to several C_i with bounds l, u and a binary connection axiom from each C_i to C with bounds l_i, u_i. In this case new bounds are calculated for component C on the left-hand side. For this computation we combine a one-to-many connection with all existing binary connections between the current component and the

components on the many-side. In more detail, we take the cardinalities from the one-to-many axiom in direction to the set and the cardinalities of the binary connection axioms in direction to the current component and compute bounds analogously to a simple binary connection as outlined above. Here we get the following inequalities:

$$\sum_i l_i * |\mathcal{C}_i| \leq u * |\mathcal{C}| \text{ and } l * |\mathcal{C}| \leq \sum_i u_i * |\mathcal{C}_i|, \tag{3.19}$$

as each element of $\mathcal{C}_i$ has to be connected to at least l_i elements of $\mathcal{C}$, whereas each of the latter can be connected to at most u elements of $\bigcup \mathcal{C}_i$. This results in the following bounds for $\mathcal{C}$:

$$\left\lceil \frac{\sum_i l_i * \lfloor |\mathcal{C}_i| \rfloor}{u} \right\rceil \leq |\mathcal{C}| \leq \left\lfloor \frac{\sum_i u_i * \lceil |\mathcal{C}_i| \rceil}{l} \right\rfloor \tag{3.20}$$

Bounds for exclusive OR one-to-many connections In the case of an exclusive disjunction in the one-to-many axiom (3.5) each element of $\mathcal{C}$ can be connected to elements of one of the $\mathcal{C}_i$ only. Let x_i denote the number of times some element of $\mathcal{C}$ uses $\mathcal{C}_i$ for its connections. Then we get for all i:

$$\sum_i x_i = |\mathcal{C}| \text{ with } l_i * \lfloor\, |\mathcal{C}_i| \,\rfloor \leq x_i * u \text{ and } x_i * l \leq u_i * \lceil |\mathcal{C}_i| \rceil \tag{3.21}$$

We observe that for both formulas (3.19) and (3.21) we need both $l > 0$ and all the $\mathcal{C}_i$ to be finitely bounded in order to derive a finite bound on $\mathcal{C}$.

Bounds for general one-to-many connections Next consider a general one-to-many axiom (3.6) and let l_j, u_j denote the lower and upper bounds in the binary connection axiom in the direction from C_j to C. Again, denote by x_i the number of times case i

applies. Then we have for all i:

$$\sum_i x_i = |\mathcal{C}| \text{ with } l_j * |\mathcal{C}_j| \leq \sum_i x_i * n_{i_j} \leq u_j * |\mathcal{C}_j|, \text{ all } j \tag{3.22}$$

The formulas above refine the bounds on the domain of components for single local connections. However, if the domain size of one component is updated, then the domain size of other components may have to be updated again because of this changes.

3.5.2 Globally Bounding Component Numbers

We formalize these local interactions between different component types in two ways, via a so-called configuration graph and via a set of Horn formulas. A configuration graph is a directed and-or-graph where the different component types are the vertices. An edge from C_1 to C_2 means C_1 can be finitely bounded if C_2 is; an and-edge from C to several C_i means C can be finitely bounded if all of the C_i are. The notion of a path in such a graph is the natural tree-like generalization of a path in a directed graph.

If we have a binary connection axiom (3.1) with $l_2 > 0$ we include an edge from C_2 to C_1. For one-to-many axioms (3.4) and (3.5) we include an and-edge from C to all C_i if $l > 0$. If we have a general one-to-many axiom (3.6) we include an and-edge from C to all C_j if there is no disjunct such that all $n_{i_j} = 0$ in the one-to-many axiom.

A configuration graph maps in a very natural way to a set of Horn clauses: Each component type becomes a propositional letter. For an edge from C_1 to C_2 include the clause $C_2 \Rightarrow C_1$; for an and-edge from C_1 to some C_i include $(\bigwedge_i C_i) \Rightarrow C_1$.

Satisfiability for Horn formulas can be checked efficiently with the well-known *marking algorithm* [Dowling and Gallier, 1984], mimicking unit resolution for Horn clauses: It repeatedly marks those heads of clauses whose literals in the clause body are all marked.

From this it follows that in linear time it is possible to decide whether user-defined input components suffice to make the configuration problem finite: Initially mark all input components and run the standard Horn algorithm. Now all components are marked iff the problem is finite, meaning that in all models of the specification all component sets

have finite cardinality. Thus, we have proven the following:

Proposition 1 (Finiteness of configurations)**.** *It can be decided in linear time whether a given configuration problem is finite.*

Observe that this is a stronger result than the one presented in [Aschinger et al., 2011b]: Whenever the algorithm returns "no" the model can be made infinite by adding components that are not connected to other components.

Finding smallest sets of "input" components

If the user-defined input components do not make the problem finite we might want to recommend a smallest fix. This amounts to the following problem: Given a directed graph, find some smallest set $\mathcal{S}$ of vertices such that for every vertex there is a path ending in some vertex in $\mathcal{S}$ or the vertex is in $\mathcal{S}$ already. If the graph is acyclic taking all sinks suffices. If there are only binary connections we can contract all cycles and then take all sinks in the resulting graph in $\mathcal{O}(\text{NumberOfComponentTypes} + \text{NumberOfAxioms})$; this set is a unique representation of all cardinality-minimal sets of components that if input make the problem finite.

If there are cycles and one-to-many connections there no longer is such a unique set. We can still find all inclusion-minimal such sets, again using the Horn algorithm, as follows. Let Φ be a set of definite Horn clauses, obtained as above from a configuration graph. We first mark all variables corresponding to sinks in the graph and put them on a list `ilist`, since these will have to be input components in all finite models. Then we run the marking algorithm. If now all components are marked we output `ilist` and are done. Otherwise we call a recursive procedure `enum`. It uses on the one hand the marking algorithm from Horn logic to mark variables with 1, but additionally marks certain variables with 0 (meaning they are not chosen as input components). More precisely the procedure works as follows:

1. Let x_1 be the smallest non-marked variable in Φ. Mark x_1 with 1 and put it on `ilist`, i.e., pick x_1 to be an input component.
2. Run the marking algorithm.

3. If now all variables are marked 1 then output `ilist`, otherwise recursively call `enum`. (Note that since x_1 is marked the number of unmarked variables has decreased, but is still nonempty.)
4. Mark x_1 with 0, i.e., try x_1 not to be an input component.
5. Determine if the configuration problem can actually be made finite without picking x_1 as input component. (This test can be performed by setting all the still unmarked variables to 1, hypothetically running the marking algorithm and checking if in this way all variables will receive mark "1".) If yes, then recursively call `enum`. (Note that since x_1 is marked the number of unmarked variables has decreased, but is still nonempty.)

Note that every time `enum` is called, the following two invariants hold: First, the problem can be made finite by making a subset of the unmarked variables input components. Second, by making all variables on `ilist` input components, all components corresponding to variables marked by 1 will be finite.

Also note that every time `enum` is called, we will output one successful configuration after a number of steps that is polynomial in the number of variables, since in the worst case we will choose all remaining (unmarked) variables as input components. Such algorithms are called enumeration algorithms with polynomial delay [Johnson et al., 1988]. We remark that the run-time of such an enumeration algorithm is bounded by the number of output words times some polynomial, which is the best notion of efficiency we can hope for in this context. Hence we conclude:

Proposition 2 (Enumerating inclusion-minimal sets of inputs)**.** *There is a polynomial-delay algorithm that enumerates all inclusion-minimal sets of components that suffice to make the configuration problem finite.*

Note that there may be exponentially many such inclusion-minimal sets. Finding sets of input components that are of minimal cardinality turns out to be harder:

Proposition 3 (Cardinality-minimal sets of inputs)**.** *The problem to decide whether there is a set of components of size at most k that suffice to make the configuration problem finite is* NP*-complete.*

Proof sketch. The problem to decide if there is a key of size at most a given integer for a database under functional dependencies is NP-complete [Lucchesi and Osborn, 1978]. A subset K of the database attributes A is a key if K and the functional dependencies determine all of A. Logically this problem can be expressed as follows: The attributes A become atomic propositions $\mathcal{A}$. A functional dependency $C \to B$ becomes an implication $(\bigwedge \mathcal{C}) \Rightarrow (\bigwedge \mathcal{B})$; i.e. it can be expressed as Horn clauses. This proves hardness of our problem. Membership in NP follows by the straightforward approach to guess and verify a set of k input components that make the problem finite. □

We may assume that in practice the user incrementally adds input components to the problem until it becomes finite. Hence inclusion-minimal sets of inputs are of greater practical relevance.

Results similar to ours have independently been obtained in a different context, formal concept analysis, by Hermann and Sertkaya in [Hermann and Sertkaya, 2008].

3.5.3 Computing Bounds on Component Numbers

Given that the problem is finite we wish to compute bounds on the number of components needed. We observe that the local conditions (3.17), (3.19), (3.21) and (3.22) can naturally be expressed in integer programming. Hence lower and upper bounds can be computed by solving two integer programs per generated component. On the other hand, we can reduce e.g. the subset sum problem to a LoCo problem giving rise to condition (3.22) and we have:

Proposition 4 (Bounds computation is NP-hard)**.** *Computing lower and upper bounds on the number of components needed to solve a configuration problem in* LoCo *is* NP*-hard.*

Proof. In the subset sum problem we are given a finite set $A = \{1, \ldots, m\}$, a positive integer size $s(a)$ for each $a \in A$ and a positive integer B [Garey and Johnson, 1979]. The problem is to determine whether there is $A' \subseteq A$ such that $\sum_{a \in A'} = B$. For the reduction all we need is a LoCo axiomatization containing a component type $\mathcal{C}_j$ for every $a \in A$ and giving rise to condition (3.22)

$$\sum_i x_i = |\mathcal{C}| \text{ with } l_j * |\mathcal{C}_j| \leq \sum_i x_i * n_{i_j} \leq u_j * |\mathcal{C}_j|, \text{ all } j$$

such that $i = j = |A|$, $l_j = u_j = B$ and $|\mathcal{C}_j| = 1$ for all j as well as $1 \leq |\mathcal{C}| \leq |A|$. Finally, for each $a \in A$, let there be one corresponding disjunct i such that in that disjunct $n_{i_j} = s(a)$ for $j = a$ and $n_{i_j} = 0$ for $a \neq j$. □

In [Feinerer, 2013] it has been shown that computing lower bounds for problems containing only binary connections or basic one-to-many connections can be solved in polynomial time. The techniques used, however, do not extend to computing upper bounds or to one-to-many connection axioms of the form (3.22).

But just how tight are the bounds that we compute? The best we can hope for is that the bounds are tight for LoCo axiomatizations containing no constraints in the connection axioms, no partial configurations and also no connection generating rules. That is the axiomatization basically consists of connection axioms, the component catalogue and a specification how many input components are to be used. Unfortunately, not even in this case the bounds are tight. Assume there are two components C_1 and C_2, the former an input and the latter a generated one. Further let each C_1 be connected to at least two C_2 and each C_2 be connected to at most two C_1. If $|C_1| = 1$ by $2 * |C_1| \leq 2 * |C_2|$ we obtain a lower bound of one on $|C_2|$ — but clearly this should be two. In general, for a binary connection, this kind of error occurs when $|C_2| < l_1$ and $|C_1| > 0$ after solving $l_1 * |C_1| \leq u_2 * |C_2|$ — cf. [Feinerer, 2013] where it is proposed to fix the problem by imposing the constraint $|C_1| > 0 \Rightarrow |C_2| \geq l_1$. We generalize the idea to LoCo's one-to-many axioms and obtain the result below.

Proposition 5 (TIGHTNESS OF BOUNDS)**.** *Assume given a* LoCo *axiomatization containing no constraints on the connection axioms, no partial configurations and no connection generating rules. Then the lower and upper bounds computed are tight if the integer programming solutions obtained satisfy the following additional conditions:*

- $|\mathcal{C}_1| > 0 \Rightarrow |\mathcal{C}_2| \geq l_1$ *for every binary connection axiom (3.1);*
- $|\mathcal{C}| > 0 \Rightarrow \sum_i |\mathcal{C}_i| \geq l_1$ *for every one-to-many connection axiom (3.4);*

- $|\mathcal{C}| > 0 \Rightarrow (\bigvee_i |\mathcal{C}_i| \geq l_1)$ *for every exclusive-or one-to-many connection axiom (3.5); and*
- $|\mathcal{C}| > 0 \Rightarrow (\bigvee_i (\bigwedge_j |\mathcal{C}_j| \geq n_{i_j}))$ *for every one-to-many connection axiom of the form (3.6).*

Proof Sketch. We observe that in the absence of constraints on the connection axioms, partial configurations and connection generating rules model finding reduces to finding component sets of a suitable size as well as suitable interconnections. We then observe that the linear inequalities are derived from the minimum and maximum number of connections between the respective sets of components, but not the minimum and maximum cardinality of those sets. In the case of upper bounds the maximum number of connections into a set is also an upper bound on that set's cardinality. However, as illustrated by the above example, for lower bounds this analogy does not hold.

So assume given a solution to the integer program. Then consider a binary connection axiom from $\mathcal{C}_1$ to $\mathcal{C}_2$. The conditions

$$l_1 * |\mathcal{C}_1| \leq u_2 * |\mathcal{C}_2| \text{ and } l_2 * |\mathcal{C}_2| \leq u_1 * |\mathcal{C}_1|,$$
$$|\mathcal{C}_1| > 0 \Rightarrow |\mathcal{C}_2| \geq l_1 \text{ as well as } |\mathcal{C}_2| > 0 \Rightarrow |\mathcal{C}_1| \geq l_2$$

jointly guarantee that we can find valid connections: For each $\mathcal{C}_1$ there are at least l_1 $\mathcal{C}_2$ to connect it to and the overall number of $\mathcal{C}_2$ is at least large enough to connect all the $\mathcal{C}_1$ (and analogously in the other direction). In general, the linear "local bounds" inequalities guarantee that there are enough components from the right hand side to connect all the components from the component type on the left hand side of a connection axiom. The additional conditions satisfied by the solution to the integer program guarantee that for each left hand side component there are enough different right hand side components to connect it to: For a one-to-many connection axiom of the form (3.4) the overall number of right hand side components is large enough; for an exclusive-or one-to-many connection axiom of the form (3.5) there is a sufficient number of at least one of the right hand side components; and for a one-to-many connection axiom of the form (3.6) there are sufficient numbers of the right hand side components so that at least one of the cases applies. □

Example 18. *We want to show the interplay of connection axioms in terms of bounds computation on a snapshot of our running example, distinguishing between two types of things A and B that both have to be stored in cabinets. Things are input components while Cabinets are of type generated with the aim of their number being minimised. We take the binary connections from our running example in figure 1.1 and set the constants* cMaxNrTA *and* cMaxNrTB *to 3 and 5 respectively. Assume having an instance with 10 Things of each kind, connection* ThingA-Bin *gives a lower bound of 4 and connection* ThingB-Bin *gives a lower bound of 2 for component Cabinet using the lower bound computations defined in formula 3.18. We take the maximum of all computed lower bounds, hence the lower bound for Cabinet is 4.*

Since in the binary connections we state that the cardinality lower bounds from Cabinet to both types of Things is zero, a model could potentially contain an infinite number of empty cabinets. This results in the fact that we can't compute an upper bound for Cabinet using solely the binary connections and would furthermore violate the finite model requirement. In order to express that for a Cabinet to exist it needs to have at least one Thing in it, we define a one-to-many connection between Cabinet and the set of Things exactly as in example 4. It is sufficient to only define a lower bound for this connection and in conjunction with the binary connections we can compute an upper bound of 20 for Cabinet by using the computation of formula 3.20, a scenario which would occur if every Thing would be put in a separate Bin.

Next let us consider the following question: Given a LoCo axiomatization, just how many components can there be in the worst case? First let us point out that cycles in the configuration graph can only lead to a decrease, but not to an increase of the upper bounds. Then assume we have $2n$ binary connection axioms forming a path $(C_1, C_2, \ldots, C_n)$ in the configuration graph, with C_1 the only input component. Then, for $1 \leq i < n$, let each C_i be connected to exactly two C_{i+1} and each C_{i+1} be connected to exactly one C_i. As this describes a complete binary tree with each component type forming one level of the tree there will be 2^n instances of C_n at the leaf level, i.e. exponentially many, cf. Figure 3.1.

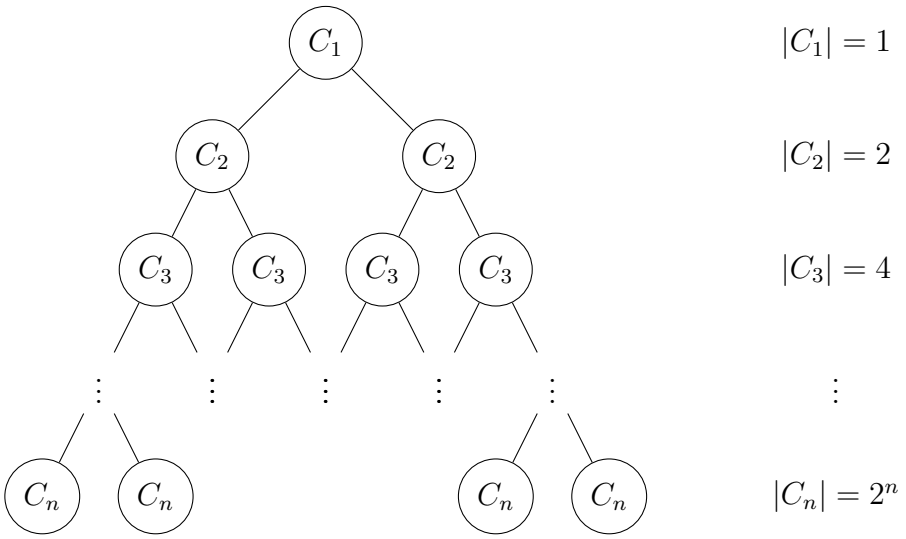


Figure 3.1: A full binary tree rooted at an input component of type C_1

3.6 The Complexity of Deciding LoCo Satisfiability

We now turn to the computational complexity of determining whether a LoCo axiomatization admits a model or, equivalently, whether there exists a configuration satisfying the requirements. As there can be configurations containing exponentially many components the usual technique for showing membership in NP ("guess and check") does not work for LoCo satisfiability. Still in [Aschinger et al., 2012] we expressed the hope that there might be some workaround such that the question can nevertheless be decided in NP. As the following result shows this only holds in the unlikely case of NP = ExpTime.

Proposition 6 (Deciding LoCo Satisfiability is ExpTime-complete)**.** *Assume given a* LoCo *domain axiomatization plus instance knowledge with an intensional component catalogue. Then it is* ExpTime*-complete to decide whether there exists a model satisfying the axioms.*

Proof. We first show membership in ExpTime. Deciding finiteness, identifying suitable sets of input components and computing bounds on the number of components clearly can all be done in ExpTime. We may hence assume that the number of instances for all component types is finitely bounded. Observe that no component type may have more

than exponentially many instances and that all attributes have finite ranges. In particular, there are at most exponentially many different combinations of attribute values that satisfy the component catalogue. Hence in ExpTime we can generate:

- for each component type $\mathcal{C}_i$ all possible component sets within the size bounds,
- all possible combinations of these sets , and
- all possible extensions of the C_i_C_j relations.

Finally we check whether some combination of the different possible component connections and the different possible component sets is a model of the axioms.

Next we show hardness, by reducing an APSpace Turing machine to a LoCo axiomatization; by [Chandra et al., 1981] this suffices to show ExpTime hardness. The idea is to encode the machine configurations into components (all of the same type $\mathcal{C}_{TM}$). Transitions are then encoded as connections between the components. We need to have $\mathcal{O}(2^{p(n)})$ components at our disposal. This is achieved by repeating the construction sketched immediately below proposition 4 using $p(n)$ many component types and connection axioms. For each cell of the Turing machine we introduce an attribute. Hence the components are of the form $C_{TM}(id, b, q, t_1, \ldots, t_{p(n)})$. Here b denotes the position of the head and q indicates the machine state (including whether we are in an existential, universal, accepting or rejecting state). The t_i are the tape cells; the respective attribute values range over the tape alphabet. We may assume that the Turing machine terminates after exactly $2^{p(n)}$ steps and that component identifiers are numbered starting from "1". The initial state may then be encoded as $C_{TM}(1, b, Q_0, \_, \ldots, \_)$. We can now rule out rejecting configurations of the Turing machine by using LoCo's partial configurations: $(\forall)\, \neg C_{TM}(id, b, \text{reject}, t_1, \ldots, t_{p(n)})$. We still need to encode the machine's transitions: For this we use a binary connection axiom of the form (3.3). We assume that in the universal states each configuration has exactly two successors whereas in the existential states there is exactly one successor.

$$
\begin{array}{l}
(\forall\, id, b, q, \vec{t_i})\, C_{TM}(id, b, q, t_1, \ldots, t_{p(n)}) \Rightarrow \\
\big[\ (\\
\qquad [\, (\exists^1_1\, \hat{id})\ C(\hat{id}, \hat{b}, \hat{q}, \hat{t_1}, \ldots, \hat{t}_{p(n)}) \wedge \phi_1 \,] \vee \\
\qquad [\, (\exists^2_2\, \hat{id})\ C(\hat{id}, \hat{b}, \hat{q}, \hat{t_1}, \ldots, \hat{t}_{p(n)}) \wedge \phi_2 \,] \ ) \wedge \\
\quad [\, \hat{id} = id + 1 \,] \\
\big]
\end{array}
$$

The formulas $\phi_1 = \bigvee \psi$ and $\phi_2 = \bigvee \psi$ contain one disjunct ψ per entry in the Turing machine's transition table; ϕ_1 is for existential transitions and ϕ_2 for universal ones. For any transition leading from state Q to state $\hat{Q}$ and replacing the symbol T with $\hat{T}$ hence ψ looks as follows, with B the current and $\hat{B}$ the new head position:

$$
\begin{array}{l}
\displaystyle\bigwedge_B \ \big[\, (b = B \wedge t_b = T \wedge q = Q) \Rightarrow \\
\qquad\qquad (\hat{b} = \hat{B} \wedge \hat{t_b} = \hat{T} \wedge \hat{q} = \hat{Q} \wedge \displaystyle\bigwedge_{i \neq B} t_i = \hat{t_i}) \,\big]
\end{array}
$$

Given this basic setup it is straightforward to complete the definition of the LoCo axiomatization in such a way that the APSpace Turing machine accepts if and only if the corresponding LoCo axiomatization has a model: We stipulate that there are no connection axioms beyond the binary connection from C_{TM} to itself just sketched and the binary connections used in the construction below proposition 4 in order to obtain $2^{p(n)}$ instances of C_{TM}. For the latter we may assume that the respective component types have no attributes beyond their identifiers and hence that there are no ϕ-constraints in the connection axioms. Next, we may assume that the component catalogue neither specifies nor rules out any attribute combinations for C_{TM}. Finally, the partial configuration used in the LoCo axiomatization shall consist only of the Turing machines initial state and the axiom ruling out rejecting configurations. □

Discussion

It is well worth pointing out that the above proof does not work if we use an extensional component catalogue as in this case all the reachable configurations of the Turing machine have to be listed explicitly. On the other hand it is interesting to observe that the proof uses only a very basic subset of the rich LoCo language: Binary connection axioms with very simple ϕ-constraints and partial configurations suffice. So, in some sense, the more expressive connection axioms of LoCo that admit much more natural problem formulations come for free.

Chapter 4

LoCo Input Language

In order to put the LoCo axiomatization into practice we created the LoCo input language (LIL). A major design requirement was the ability to represent the configuration knowledge in standard ASCII text format. Nevertheless, as will be shown in this chapter, the basic structure of this language is still strongly oriented on the original axiomatization. The LoCo input language is separated into domain and instance knowledge. The domain knowledge defines all used elements in the model, i.e. constants, components, attributes, connection predicates, their associated connection axiom rules as well as consistency rules for non-local constraints. On the contrary the instance knowledge then feeds the model with the necessary data although in the LIL language there is no mutual exclusion and these areas overlap to some extent as we will see. Generally the static knowledge concerning the problem domain is defined in the LIL domain knowledge, whereas the LIL instance knowledge contains the dynamic knowledge related to a specific problem instance. The supported elements for describing the domain and the input knowledge will be presented in detail and differences to the original axiomatization will be pointed out.

4.1 Basic elements

4.1.1 Constants

Constants can be defined either on the domain or on the instance knowledge level but have to be declared on the domain level. Their definitions are pretty straight-forward. Currently the prototype supports only integer constants although if needed the system is easily expandable. Listing 4.1 shows the constants for our running example, starting with values for the maximum height and width of a house component (lines 1-2) followed by the maximum allowed sizes for things A and B (lines 3-4). Line 5 declares a constant cMaxNrTB representing the upper bound of how many B things can be put in a cabinet. Its value is left open on the domain level, meaning it needs to be instantiated on the input level and because of this can vary for any given problem instance. Declarations without a value assignment are generally not allowed on the instance level. Next to a standard value assignment constants can also be defined by the use of arithmetic expressions as shown in lines 7 and 8. Line 8 defines the maximum size of a cabinet for storing things of type B with the value set to the product of the two constants cMaxNrTB and maxSizeTB. Due to the fact that cMaxNrTB gets its value not until the instance level the same also holds for cMaxSizeTB. The maximum number of allowed thingA components for each cabinet in line 7 on the other hand is defined via a value constant and so the value for cMaxSizeTA is already defined on the domain level.

4.1.2 Component and Attribute definitions

Every component is either defined as an input component (`IC`) or as a generated component (`GC`). On the domain knowledge level components can also be left undefined (`UC`) but still be used the same way as `IC` or `GC` components in rules and constraints. In such a case their type needs to be stated explicitly later on the instance level where no undefined components are allowed anymore. This feature makes it easy to quickly change the problem structure in terms of which components are fixed and for which bounds have to be computed. The internal ID attribute serving as the component key does not have to be defined explicitly and is always implicitly part of every component definition. For

```
1  const hMaxHeight = 8.
2  const hMaxWidth = 5.
3  const maxSizeTA = 10.
4  const maxSizeTB = 12.
5  const cMaxNrTA.
6  const cMaxNrTB.
7  const cMaxSizeTA = 5 * maxSizeTA.
8  const cMaxSizeTB = cMaxNrTB * maxSizeTB.
9  const cMaxSize = cMaxSizeTA + cMaxSizeTB.
10 const rMaxPos = 4.
```

Listing 4.1: LoCo constants

attributes we currently support the data types Boolean, Integer and Enum. All attribute values are clearly defined, i.e. open intervals or undefined bounds are not allowed. Integer attributes are either bounded by range, i.e. by lower and upper bound, or by enumeration which allows to leave "holes" in the domain. The domain of a Boolean attribute is automatically set to TRUE and FALSE, represented internally by 1 and 0. Enum attributes are basically represented by a set of Strings, i.e. {red,blue,green} for attribute color of component cabinet.

Listing 4.2 shows the component definitions for our running example. Every component definition starts with the component name followed by a colon. Next we specify the component type, i.e. `IC` for input, `GC` for generated or `UC` for (currently) undefined as previously mentioned. This is followed up by a list of attributes encompassed in parentheses. Attribute definitions are optional and in case of no explicitly specified attributes the only component attribute is the internal ID. An example for this empty case is component person in line 1. In the second line we define input component house which has 2 Integer attributes height and width. Both of them are bounded by range with lower bounds zero and respective constants from listing 4.1 as upper bounds. The thingA component in line 3 contains some Boolean attributes to specify properties such as big and dirty.

Notice that apart from the upper bound for attribute size the component definition for thingB is identical to thingA. This scenario illustrates the potential advantage of some form of classification-based inheritance supporting `is-a` relationships like in taxonomic

```
1  person:IC ().
2  house:IC(height:INTEGER[0..hMaxHeight];width:INTEGER[0..hMaxWidth]).
3  thingA:IC(size:INTEGER[0..maxSizeTA];big:BOOL;dirty:BOOL;forUpper:BOOL;
4      minFloor:INTEGER[0..hMaxHeight];maxFloor:INTEGER[0..hMaxHeight]).
5  thingB:IC(size:INTEGER[0..maxSizeTB];big:BOOL;dirty:BOOL;forUpper:BOOL;
6      minFloor:INTEGER[0..hMaxHeight];maxFloor:INTEGER[0..hMaxHeight]).
7  cabinet:GC(size:INTEGER[0..cMaxSize];dirty:BOOL;big:BOOL;top:BOOL;
8      color:ENUM[red,blue,green]).
9  position:GC(nr:INTEGER[0,1,2,3]).
10 room:GC(size:INTEGER[0.."2*cMaxSize*rMaxPos"];
11     floor:INTEGER[0..hMaxHeight]; pos:INTEGER[0..hMaxWidth]).
```

Listing 4.2: Definition of LoCo components and attributes

hierarchies. For our running example, we would want to define an abstract component thing containing all common attributes of the A and B types with thingA and thingB being derived components each defining their own size attribute. This feature is not yet implemented in the prototype but definitely considered as part of suggested future work. One can find more on this topic in the closing chapter of this thesis.

The nr attribute of component position in line 9 is an example for an enumerated Integer domain. Although in the given scenario the value enumeration isn't necessarily needed and could be replaced by a simple range bound [0..3], we wanted to include at least one example to show the proper syntax for this use case. In general there are many common application scenarios requiring the explicit modelling of "holes" in an attribute domain interval though and this feature eases the representation of problem knowledge sufficiently. For example, let's consider a scenario where a component can only be connected to another component at certain port numbers. Instead of listing the whole domain and restricting the usage of forbidden ports by using constraints, we may define an additional attribute allowedPorts for the component to be connected, containing the enumeration of valid port numbers as its domain. The Enum attribute color in line 8 is another example where the usage of an enumeration clearly simplifies the representation of knowledge.

Next to using constants and numbers we can also define attribute bounds via arbitrary arithmetic expressions. The upper bound for attribute size of component room is an example for this situation. It is calculated by taking the maximum cabinet size (cMaxSize), multiplying it with the maximum number of positions in a room (rMaxPos) and further multiplying this by 2 since up to 2 cabinets can be put on every position.

The LoCo Input Language also supports the option to specify bounds on the number of allowed instances for a component on the domain level. Usually component bounds are set on the instance level but for some components it might be useful to predefine bounds in case the number of instances stays constant, e.g. due to technical restrictions. The way to achieve this effect is by putting lower and upper bounds in square brackets between the component type and the component attributes when defining a component type, like so:

```
1 thingA:IC[15,25]( ... ).
2 thingB:IC[?,25]( ... ).
3 cabinet:GC[1,10]( ... ).
4 room:GC[3,?]( ... ).
```

Listing 4.3: Component definitions with bounds

Component bounds in this context can be either defined for input or for generated components although with different semantics. In case of an input component the upper bound is not only a bound but defines the exact number of instances. The lower bound has no effect since there will always be upper bound many instances. It is possible to only define a lower or an upper bound by using the question mark placeholder for unspecified bounds. For example thingB in line 2 has no lower bound but since it is an input component its bound definition is identical to the one for thingA. For a generated component the bounds set the minimum and maximum number respectively of allowed generated components. In case both lower and upper bounds are specified like for a cabinet in line 3, these manual bounds hold and the component is excluded from getting its values by the automatic bounds computation. Line 4 expresses the fact that there have to be at least 3 instances of generated component room while the bounds computation algorithm will determine the corresponding upper bound later on in the process.

4.2 Connection Rules

4.2.1 Binary connections

Standard Binary Connection

We investigate the syntactic differences between the axiomatization and the input language based on the abstract form of a binary connection. To recapitulate, the axiomatized version from chapter 3 looks like this:

$$(\forall\, id_1, \vec{x})\ comp_1(id_1, \vec{x}) \Rightarrow \\ (\exists_{l_1}^{u_1}\, id_2)\ [\, comp_1\_comp_2(id_1, id_2) \wedge comp_2(id_2, \vec{y}) \wedge \phi(id_1, id_2, \vec{x}, \vec{y})\,] \tag{4.1}$$

The equivalent syntactic structure in the LoCo input language has the following form:

$$\text{(FORALL C1) comp1(C1)} \Rightarrow \\ \text{(EXISTS [l1,u1] C2) [comp1\_comp2(C1,C2) \&\& comp2(C2) \&\& } \phi\text{(C1, C2)]} \tag{4.2}$$

It can be clearly seen that the text-based syntax of the LIL format matches strongly with the original axiomatization. The specific logical symbols get replaced by appropriate keywords, e.g. the universal quantification $\forall$ is matched to FORALL, existential quantification with bounds $\exists_{l_1}^{u_1}$ to EXISTS[l1,u1], etc. Also logical connectives get replaced by textual representations. Binary connectives $\wedge$ and $\vee$ become && and || respectively, the unary negation connective $\neg$ becomes ! and so on s.t. all symbols have meaningful textual equivalents. Table 4.1 summarizes the mapping.

Component variables C1 and C2 are used to address the component instance identifiers id_1 and id_2 from the axiomatization but in addition to that also represent its associated attribute vectors $\vec{x}$ and $\vec{y}$. From this it follows that component attributes in the LIL format don't get quantified explicitly like in the axiomatization but implicitly instead via the component variables. They can be addressed via *dot-notation* in a similar way as in the standard object-oriented paradigm, i.e. via <*Component*>.<*Attribute*>. For

Logical symbol	Axiomatization	LoCo Input Language
Universal quantifier	$\forall$	FORALL
Existential quantifier	$\exists$	EXISTS
Conjunction	$\wedge$	&&
Disjunction	$\vee$	||
Negation	$\neg$	!
Implication	$\Longrightarrow$	=>
Equality	$\Longleftrightarrow$	<=>

Table 4.1: Mapping of language elements

a practical example the general format of a ϕ subformula in 4.2 gets replaced by a constraint expression. More on this together with different ways of addressing component attributes will be discussed on snapshots of our running example below.

We now illustrate the mapping of a standard binary axiom on a transformation of the connection from thingA to cabinet as already shown in Example 2 together with the reverse direction shown in Example 19.

Example 19. *In the House Problem each cabinet contains between* zero *and* cMaxNrTA *things of type A whereas for each cabinet the sum of the size of all big things is not allowed to be greater than five; moreover, a cabinet can only be put on a top position if all things in it are suitable for being placed on top:*

$$
\begin{aligned}
&(\forall\, id_C, \vec{attr}_C)\ \ cab(id_C, \vec{attr}_C) \Rightarrow \\
&\quad (\exists_0^{cMaxNrTA}\, id_{TA})\ \ thingA\_Cab(id_{TA}, id_C) \wedge thingA(id_{TA}, \vec{attr}_{TA}) \ \wedge \\
&\quad\quad [\, \sum (\, size_{TA}\, [big_{TA} = TRUE]\,) \leq 5 \wedge (\neg top_C \vee forUpper_{TA})\,]
\end{aligned}
$$

The LIL representations for Examples 2 and 19 are depicted in formulas 4.3 and 4.4. Note that we can use either numbers or constants for expressing bounds of existential counting quantifiers. For example in formula 4.4 the upper bound for thingA is defined by constant cMaxNrTA.

```
(FORALL TA) thingA(TA) ⇒                                                    (4.3)
      (EXISTS [1,1] C) thingA_cab(TA,C) && cab(C)  &&
        [ (C.big && TA.big) || !(TA.big) ].

(FORALL C) cab(C) ⇒                                                        (4.4)
      (EXISTS [0,cMaxNrTA] TA) thingA_cab(TA,C) && thingA(TA)  &&
        [ SUM(TA.size [ TA.big == TRUE ] ) <= 5  &&
          ( !(C.top) || TA.forUpper == TRUE) ].
```

Both binary connections respectively contain ϕ constraint parts. The one in formula 4.3 is a simple comparison of Boolean attributes, i.e. we check that in case a thing is marked as big then it can only be put in a cabinet that is also big. As mentioned before attributes like big or size can be accessed via the component variables in the constraint part, e.g. we can write TA.size to get attribute size of component thingA which is identified by component variable TA.

The constraint of the reverse direction in formula 4.4 uses a summation aggregate to determine the total size of all things in the cabinet. Aggregate functions can have optional nested constraints. In our case $[\,TA.big == TRUE\,]$ restricts things to be taken into consideration to those having attribute big set to TRUE. Note that while it is possible to use any component variable and its related attributes, in practice it only makes sense to use aggregate functions in connection with existentially quantified variables. For example in formula 4.4 component variable TA is existentially quantified and so an expression like $TA.size$ addresses all instances of component thingA belonging to this relation. An aggregate involving attributes of universally quantified component variable C would only make a computation over a single instance of a cabinet.

Unfolded Binary Connections

The unfolded version of a binary connection also has its representative in the LIL format. Its basic use is to achieve a higher level of granularity by allowing a direct addressing of specific component instances. Formula 4.5 shows the textual LIL representation of example 3:

```
(FORALL P) pos(P) ⇒                                                    (4.5)
      {(EXISTS [1,1] C) cab_pos(C,P) && cab(C) &&
          [ C.top == FALSE ] } ||
      {(EXISTS [2,2] C) cab_pos(C,P) && cab(C) &&
          [ !(C.big) && ( (C[1].top && !(C[2].top)) || ( !(C[1].top) && C[2].top) ) ] }.
```

The rule differs between the two cases that either 1 or 2 cabinets are placed on a position. Each case has its own ϕ sub-constraint. In the case of 1 connected cabinet attribute top needs to be set to FALSE, expressing the fact that the cabinet cannot be placed on top of another cabinet. Just like a Boolean comparison $[\,C.top == TRUE\,]$ could be equivalently expressed with reduced syntax $C.top$, also $[\,C.top == FALSE\,]$ could be formulated equivalently as $!(C.top)$ for negating Boolean attributes.

The constraint for the second case is slightly more involved and needs some additional explanation. The first conjunct $!(C.big)$ states that in the case of 2 cabinets on a position none of them can have its attribute big set to TRUE. Like in a standard binary connection constraint, addressing an attribute via $C.big$ without an index means that big has to hold for all instances related to component variable C. The second and third conjuncts express the fact that exactly one of the two cabinets needs to have the top attribute being set to TRUE such that it is placed on top of the other cabinet.

Specific component instances can be addressed directly via an index after the component identifier, e.g. $C[1].top$ for the cabinet with index 1. The order of the instances is undefined and generally of no matter, there just has to be one permutation of instances that satisfies the constraint. Take for example a case with 3 instances and a constraint

part where instances with indices 1 and 2 need to have certain properties. Then any 2 of those 3 component instances need to satisfy the requirements, i.e. one of the three instances fulfills the role of index 1 and another the role of index 2. One of them, no matter at which original position, remains "free" and won't be considered in terms of fulfilling the constraint.

We see that also in the case of unfolded binary connections the mapping from the axiomatization to its textual LIL representation is quite natural. Note that when defining an unfolded binary connection we don't need to cover the whole range from the unique lower to upper bound and are allowed to leave "holes" in the bounds, i.e. by only stating those cases for which special constraints exist. For this reason there always needs to exist a standard binary connection in addition to an unfolded one for the same direction in order to specify the whole range, e.g. the concrete example requires the following additional rule:

```
(FORALL P) pos(P) ⇒                                          (4.6)
        (EXISTS [1,2] C) cab_pos(C,P) && cab(C).
```

Alternatively to using an unfolded binary connection we could express the same matter with a standard binary connection and some aggregate functions as shown in formula 4.7. In this version the constraint part is separated into 2 disjunctive cases having either 1 or 2 cabinets on a position. This is done respectively by the use of counting aggregates $[\,COUNT(C) == n\,]$ to determine the number n of connected cabinets. In the case of one cabinet we simply form a conjunction of the aggregate with the original constraint part *!(C.top)*. The second disjunct uses an additional counting aggregate to exclude the case of 2 cabinets being placed on the same position that are both marked as top. More precisely, the aggregate $COUNT(C.top)$ returns the number of cabinets with attribute top set to TRUE and its return value has to be 1. This replaces the part of the unfolded rule where we directly address component instances. Note while this does work in the specific current case that in general an unfolded connection is more expressive and cannot be replaced with a standard binary connection, i.e. we cannot always replace a direct

addressing of component instances with other language elements like aggregates.

$$\begin{aligned}&\text{(FORALL P) pos(P)} \Rightarrow \\ &\quad \text{(EXISTS [1,2] C) cab\_pos(C,P)} \wedge \text{cab(C)} \wedge \\ &\quad\quad [\,(\,\text{COUNT(C)} == 1 \;\&\&\; !\text{(C.top)}\,) \;\; || \\ &\quad\quad\quad (\,\text{COUNT(C)} == 2 \;\&\&\; !\text{(C.big)} \;\&\&\; \text{COUNT(C.top)} == 1\,)\,].\end{aligned} \tag{4.7}$$

4.2.2 One-to-many connections

This section contains the mappings of the various LoCo one-to-many connection axioms from chapter 3. We shall see that just as for binary connections the text-based LIL syntax goes in accordance and matches strongly with the LoCo axiomatization.

Standard One-to-Many connections

We start with the LIL representation of the standard one-to-many connection from example 4 in chapter 3. This defines the relation between a cabinet and things of type A and B and is shown in formula 4.8. The lower bound of 1 ensures that there can be no empty cabinets and at least one thing of type A or B needs to be placed in it. A question mark is used as a placeholder for the upper bound of the counting quantifier, meaning the bound is unknown or left undefined. Since we stipulate that in parallel there need to exist binary connections between the component on the left-hand side and all components appearing on the right-hand side of every one-to-many connection, the binary connections of our running example define the upper cardinality bound for the one-to-many connection in this case.

$$\begin{aligned}&\text{(FORALL C) cab(C)} \Rightarrow \\ &\quad \text{(EXISTS [1,?] T) [(T=TA \&\& thingA\_cab(TA,C) \&\& thingA(TA)) ||} \\ &\qquad \text{(T=TB \&\& thingB\_cab(TB,C) \&\& thingB(TB))] \&\&} \\ &\qquad \text{[COUNT(TA.big) + COUNT(TB.big) <= 4].}\end{aligned} \tag{4.8}$$

The ϕ sub-constraint counts the total number of big things in each cabinet by using a counting aggregate function for each type of thing. In case a cabinet contains only A or B things then the other counting aggregate with no elements simply returns zero. The rule in formula 4.9 shows the resource balancing example 5:

$$\begin{aligned}&\text{(FORALL C) cab(C)} \Rightarrow \\ &\quad \text{(EXISTS [2,4] T) [(T=TA \&\& thingA\_cab(TA,C) \&\& thingA(TA)) ||} \\ &\qquad \text{(T=TB \&\& thingB\_cab(TB,C) \&\& thingB(TB))] \&\&} \\ &\qquad \text{[SUM(TA.tRes) >= SUM(TB.tRes)].}\end{aligned} \tag{4.9}$$

Exclusive-OR One-to-Many connections

Alternatively let's say there's a scenario where things of type A and B are not allowed to be put together in the same cabinet and we would need exclusive disjunction. The exclusive-or (XOR) variant in rule 4.10 stipulates that either 1 to 5 things of type A or 1 to 3 things of type B can be placed in a cabinet. An exclusive-or one-to-many connection does not support ϕ sub-constraints because since the components on the right-hand side are mutually exclusive the constraints between any of them and the component on the left-hand side can be modelled via binary connections. We represent the exclusive-OR operator $\oplus$ from the axiomatization with its textual LIL equivalent ^^ . The choice of using ^^ as the textual representation of the logical XOR connective was influenced by the common use of a single ^ for a bitwise XOR operation.

(FORALL C) cab(C) ⇒ (4.10)
[(EXISTS [1,5] TA) thingA_cab(TA,C) && thingA(TA)] ^^
[(EXISTS [1,3] TB) thingB_cab(TB,C) && thingB(TB)].

General One-to-Many connections

The highest level of granularity can be achieved with a general one-to-many rule. Similar to an unfolded binary connection this makes it possible to directly address the individual connected components. We demonstrate this on the basis of example 7 from the previous chapter. Note that the LIL version extends the original version by an additional case that groups several connections together. The mentioned case for example ranges over 4 to 7 thingA and 5 to 7 thingB instances whereas in the axiomatization each of those cases would have to be written down explicitly. By using this additional syntactic sugar the LIL format allows to reduce the total size of the formula.

(FORALL C) cab(C) ⇒ (4.11)
{ [(EXISTS [1,1] TA) thingA_cab(TA,C) && thingA(TA)] &&
[(EXISTS [1,1] TB) thingB_cab(TB,C) && thingB(TB)] &&
[thingA_thingB(TA,TB)] } ||
⋮
{ [(EXISTS [2,2] TA) thingA_cab(TA,C) && thingA(TA)] &&
[(EXISTS [3,3] TB) thingB_cab(TB,C) && thingB(TB)] &&
[TA[1].size > TB.size] } ||
⋮

{ [(EXISTS [3,3] TA) thingA_cab(TA,C) && thingA(TA)] &&
[(EXISTS [4,4] TB) thingB_cab(TB,C) && thingB(TB)] &&
[SUM(TA.size) < 5] } ||

⋮

{ [(EXISTS [4,7] TA) thingA_cab(TA,C) && thingA(TA)] &&
[(EXISTS [5,7] TB) thingB_cab(TB,C) && thingB(TB)] }.

4.3 Consistency Rules

We now look at the LIL representations of the consistency axioms. The basic syntactic elements have already been discussed in the previous section and won't be repeated at this point anymore. Similar to the connection axioms the following types of rules are strongly oriented on the original axiomatization with the same main difference of replacing first-order language elements by suitable textual equivalents.

4.3.1 Candidate Key rules

Formulas 4.12 and 4.13 show the LIL representations of the respective examples 8 and 9 of chapter 3. After quantifying all involved component variables and assigning them to a component type, the conjunction of attribute equalities forming the component key gets listed in the constraint part. While in the axiomatization the respective component id's must be identical in the consequent, here the component variables serve as a substitute for them.

(FORALL R1,R2) room(R1) && room(R2) && (4.12)
[R1.floor = R2.floor && R1.pos = R2.pos] ⇒ R1 == R2.

Formula 4.13 extends the list of attribute keys with additionally required component connections/ports.

(FORALL P1,P2,R) pos(P1) && pos(P2) && room(R) && (4.13)
[P1.nr = P2.nr && pos_room(P1,R) && pos_room(P2,R)] ⇒
P1 == P2.

4.3.2 Connection-generating rules

We now look at the LIL representation of connection-generating rules. In contrast to the LoCo axiomatization all component variables have to be explicitly quantified. The binary connection chain in the ϕ subformula of rule 4.14 establishes the connection from a person to a room component. Just like in other rule types the subformula could also contain additional attribute comparisons and arithmetic expressions. The consequent part after the implication consists solely of the connection instance to be compelled.

(FORALL PE,TA,C,PO,R) (4.14)
pers(PE) && thingA(TA) && cab(C) && pos(PO) && room(R) &&
[pers_thingA(PE,TA) && thingA_cab(TA,C) &&
cab_pos(C,PO) && pos_room(PO,R)] ⇒
room_pers(R,PE).

4.3.3 General First-Order rules

For the representation of the most general form of consistency rules we investigate the text-based LIL versions of the examples 11, 12 and 13 from chapter 3. Rule 4.15 depicts the simple example of an intensional component catalogue axiom.

```
(FORALL C,P) cab(C) && pos(P) &&                                    (4.15)
    [ cab_pos(C,P) && (P.nr == 1 || P.nr == 2) ] ⇒
      [ C.color == blue ].
```

Rule 4.16 is similar to a connection-generating rule but instead of enforcing the existence of a connection the consequent consists of a set of attribute comparisons that has to be fulfilled.

```
(FORALL TB,C,P,R) thingB(TB) && cab(C) && pos(P) && room(R) &&      (4.16)
    [ thingB_cab(TB,C) && cab_pos(C,P) && pos_room(P,R) ] ⇒
      [ TB.minFloor <= R.floor && TB.maxFloor >= R.floor ].
```

Next we look at the supplementation of GFO rules with existential quantifiers (formula 4.17). The necessary existence of a person component instance which is connected to a thing infers from the fulfilment of the antecedent. Notice that the existential counting quantifier has no bounds and we make no assumption on the number of existing persons apart from the fact that there has to exist at least one.

```
(FORALL TA,C) thingA(TA) && cab(C) &&                               (4.17)
    [ thingA_cab(TA,C) && TA.dirty == C.dirty ] ⇒
      (EXISTS P) person(P) && [ person_thingA(P,TA) ].
```

4.4 Domain Knowledge

The LoCo Domain Knowledge defines all used elements in the model, i.e. constants, components, attributes, connection predicates, their associated connection axiom rules as well as consistency axiom rules for non-local constraints. This involves basically all the language features we've discussed so far in this chapter. We won't repeat those and only present the remaining elements of domain knowledge.

Ports Next to connection predicates components can be also connected via so-called ports in order to support the component-port model. This is the commonly accepted and widely used representation pattern for (technical) configuration problems [Mittal and Frayman, 1989, Mailharro, 1998]. The representation of port connections in our model is analogous to the modelling of binary connections. Ports themselves are modelled just like components with the exception that they don't possess any attributes apart from their ID and are related to exactly one component. In a certain way one could view a port as an extension of the component it belongs to. The LIL format of example 14 for expressing the fact that every room component contains 4 position ports for connections with cabinets looks as follows:

```
(FORALL R) room(R) ⇒                                    (4.18)
        (EXISTS [4,4] P) pos_room(P,R) && pos(P).
```

Component Catalogue The domain knowledge also contains the mandatory component catalogue. By design the LoCo input language already makes sure to prevent the possibility of constructing models with infinite domains. Because of the requirement that all component attributes need to have finite bounds, the Cartesian product of the attribute domains stipulates a finite set of allowed tuples for each component type. So even without any specific catalogue knowledge the component definitions in listing 4.2 could represent an intensional component catalogue on its own. This would already be

sufficient for the demands of a valid component catalogue but could quite possibly produce a high number of potential attribute assignments. As discussed in the previous chapter we can narrow the set of possible assignments down by either an extensional listing of valid attribute tuples or by an intensional representation via adequate formulas. We first depict the LIL equivalent of example 15 from chapter 3:

$$\begin{aligned}&\text{(FORALL TA) thingA(TA) <=>} \\ &\quad \text{TA.attrVec} = \{\ (5, \text{TRUE,FALSE,FALSE}, 0, 2)\ \ || \\ &\qquad\qquad (3, \text{FALSE,FALSE,TRUE}, 1, 1)\ \ || \\ &\qquad\qquad (4, \text{FALSE,TRUE,FALSE}, 1, 3)\ \ || \\ &\qquad\qquad (\ldots)\ \}.\end{aligned} \tag{4.19}$$

The expression TA.attrVec in this context is used for accessing the attribute vector of component TA and hence attrVec is a reserved language keyword which is not allowed for a regular attribute name. For a component type with n attributes we need to have n-ary attribute vectors. Vector values are ordered in the same way as attributes at their time of definition, i.e. the second value of a vector belongs to the second attribute of the component definition. We don't consider the internal ID of the component since its value is set automatically. The right side of formula 4.19 after the logical equivalence contains the disjunction of all allowed attribute vectors for a component instance.

We have already mentioned in chapter 3 that the intensional catalogue axioms from example 16 and 17 can be merged to one expression. The LIL version of the combined formula looks as follows:

$$
\begin{aligned}
&\text{(FORALL TA) thingA(TA) <=>} \\
&\quad [\ (\text{(TA <= 3) => (TA.forUpper == TRUE)}\ )\ \&\& \\
&\quad\ (\text{(TA > 3) => (TA.forUpper == FALSE)}\ )\ \&\& \\
&\quad\ (\text{TA.minFloor <= TA.maxFloor}\ )\ \&\& \\
&\quad\ (\text{TA.size < maxSizeTA}\ ) \\
&\quad].
\end{aligned}
\tag{4.20}
$$

Since attributes already get finite bounds when they are declared, not all attributes of a component have to be considered in the catalogue axioms. They can be either (1) left unconsidered in the catalogue with their values getting restricted through some constraints or (2) be assigned later on the instance level in case their value is part of a specific problem setting.

4.5 Instance Knowledge

The LoCo instance knowledge contains the knowledge of a specific problem instance and in combination with the domain knowledge forms a complete problem model. It basically represents the data while the domain knowledge represents the structure although as we will see below these areas are not completely separated and overlap to some extent.

Constants and attributes The instance knowledge can eventually contain additional constants but definitely needs to provide value assignments for existing constants that have been declared as a placeholder in the domain knowledge. In general instance knowledge always overwrites domain knowledge, i.e. we can give pre-defined domain constants as well as attributes new values.

Component types While we are not allowed to change the existing component definitions at this point anymore we could still change its component types. Also the final subdivision of the component types needs to take place on the instance level. So if not

already set by the domain knowledge, we have to separate all components into either type *input* or type *generated*. From this it follows that components which have been declared as undefined need to get a type assignment.

The component definitions we've presented in listing 4.2 all have a clearly defined type so in this case there wouldn't be anything left to specify on the instance level. In order to show the benefits of a more flexible model let's suppose we change all types from the initial definition to `UC` (undefined) instead at the domain level. In doing so we would then have to define the types on the instance level. One advantage of this approach is to gain additional flexibility in how to use a problem model: For example, the House Problem could be quickly turned upside down and instead of creating cabinets and rooms for a given set of things we could ask the following question: Given a specified house layout with pre-defined rooms and cabinets in them, calculate the maximum possible number of things to be stored while a certain subset of things needs to satisfy additional constraints.

```
1 person:IC().
2 house:IC().
3 thingA:GC();
4 thingB:GC();
5 cabinet:IC();
6 position:IC();
7 room:GC();
```

Listing 4.4: Definition of component types for a modified House Problem

Listing 4.4 contains the type definitions for this scenario. The lists of component attributes here are empty since they are already defined as part of the original component definitions on the domain level.

Concerning the remark about additional constraints on things, the LoCo input language supports the usage of consistency rules in the instance knowledge. The following rule for example extends the problem by stating that blue cabinets can only contain small things:

(FORALL TA) thingA(TA) ⇒ (EXISTS C) cab(C) &&
[thingA_cab(TA,C) && (C.color == "BLUE" => TA.big == "FALSE")].

We have to use a GFO type of rule in this case since the existing connection rules as well as their constraints from the domain knowledge cannot be changed or extended. As pointed out before, the connection axioms and the component definitions are the elements for guaranteeing a finite problem model and so additional consistency axioms can only narrow the finite problem space further. In the worst case this would result in an unsatisfiable problem and therefore cannot lead to undecidable or infinite configurations.

Input components If not already done as part of the component definition we need to stipulate for every input component exactly how many instances exist in the model. The same procedure is also applicable for generated components although not necessary because in case of no explicit bounds the automatic bounds computation ensures the finiteness for them. Concerning input components though the bounds computation expects them to be set manually.

In order to define complete input components or partial configuration constructs of any kind we have the additional option to assign values to attributes of specific component instances. Listing 4.5 illustrates the syntax and shows a snapshot of some knowledge for input components.

In line 1 the upper bound for component person is set to two. Since person is an input component the upper bound reflects at the same time the exact number of instances. Our model furthermore contains one house instance (line 3) for which we also set its attributes height and width (lines 4-5). Attributes in this context can be addressed via dot-notation similar to how they can be accessed in ϕ constraints. In order to state which instance the attribute belongs to we list the internal ID in brackets after the component name. Internal IDs always start at 1 for the first component and will be allocated automatically in ascending order. So since there is only one house instance we address its attribute values using component ID 1. The remaining code lines below

```
1  #person = 2.
2
3  #house = 1.
4  house(1).height = 5.
5  house(1).width = 2.
6
7  #thingA = 5.
8  thingA(1).size = 3.
9  thingA(1).big = TRUE.
10 thingA(1).dirty = FALSE.
11 ...
12 thingA(2).size = 4.
13 thingA(2).big = FALSE.
14 ...
```

Listing 4.5: Input component knowledge

depict the domain closure for thingA including a couple of attribute definitions for various instances. In case bounds for a component have already been specified at the time of definition the previous existing bounds will be overwritten. Note that not all attribute values of an input component have to be specified though. The values of the remaining unspecified ones might be determined by (resource) constraints, connection structures or optimization conditions during the actual model finding process. If an attribute is not relevant for the current configuration and not influenced by any of the aforementioned language elements then it will just get a random value of its domain in a model.

Input connections Next to input components we can also define domain closures on input connections. A definition begins with the keyword `IConn` followed by the predicate name of the connection which needs to be consistent throughout the model. Encompassed in curly brackets one then specifies the set of actual connection instances. The involved component instances are identified by their internal IDs in the correct order of the predicate. Listing 4.6 shows an example input connection definition between components person and thingA.

```
1 IConn: person_thingA {
2   person_thingA(1,1).
3   person_thingA(1,2).
4   person_thingA(2,3).
5   person_thingA(2,4).
6   person_thingA(1,5).
7   ...
```

Listing 4.6: Input connections between persons and things

Partial configurations A common use case in configuration is to start the solving process from an existing partial configuration which then needs to get extended to a valid model. The solving process for this scenario is analogous to searching for a model from scratch with the main difference that a solution requires the pre-defined set of component and connection instances to be part of the configuration. In a certain way one could see the input components and connections discussed above already as some form of partial configuration. While for input elements the exact listing of all entities plus a closure on their domains is mandatory, we also want to enforce the existence of specific components and connections without ruling out the existence of additional instances of the same types, i.e. without closing their domain. This feature therefore focuses mainly on the generated components and connections for the purpose of building partial configurations. The following LIL code snippet gives some insight on how this can be achieved:

```
1 #cabinet = 7.
2 room(5).
3 room(5).size = 18.
4 room(5).floor = 2.
5 room(7).roomID = R005.
6 room(7).size = 15.
7 ...
8 cab_thingB(2,3).
```

Listing 4.7: Partial configuration knowledge

The first line of listing 4.7 shows a component bound definition for component cabinet using the same syntax as above for input components. In fact, its effect is also similar in terms of bounds by setting the upper bound to 7 but with the difference that for a generated component this does not determine how many instances of those will actually be part of a model apart from that it can't be more than 7.

The enforcement of a room component instance with ID 5 in line 2 is the LIL equivalent of a simple ground fact in the axiomatization. Referencing ID 5 implicitly requires that there need to exist at least 5 instances in total or the problem becomes unsatisfiable because internal IDs for components start to get counted from 1 upwards in ascending order. Furthermore we set some attribute values for this component instance (lines 3-4). The assignment of an attribute value for a room component with index 5 implicitly implies that the specified component instance has to exist. This means that either line 3 or 4 would make line 2 redundant but for better readability and maintenance though it is still recommended to also define the component instance explicitly.

Let's imagine a scenario where the structure of the house plan provides a specific set of identifier keys for rooms which should be used instead of the automatic IDs. The best way to handle this is to define an additional attribute for a room representing these manual IDs. Of course it doesn't necessarily have to be a number and could also be alphanumeric in form of an enumeration attribute like roomID in line 5. We could now address the specific instance by this attribute instead of the ID in connection or consistency rules.

Next to components partial configurations can also contain specific connections as demonstrated in line 8. The mentioned component instances of the connection need to be part of a valid solution, i.e. the cabinet instance with ID 2 and the thingB instance with ID 3 need to be among the selected component instances of a model.

We can also exclude certain instances of components and connections from being part of a model. This is simply done by putting an exclamation mark representing unary negation as a prefix. The exclusion of a component instance like room with ID 4 in line 1 prevents it also from being part of any connection. This definition of no-go entities of course only makes sense for generated components and for connections that are not of

type input.

```
1 !room(4).
2 !cab_thingB(2,4).
3 ...
```

Listing 4.8: Excluding components and connection instances

While partial configurations are primarily supposed to be defined on the instance level, it is possible to define component and connection instances also on the domain level in order to make them static for all instances. A possible scenario for this could be that due to the technical specification some specific instances of type A should be included in every model and hence should not be defined on the instance level.

Chapter 5

Prototypical Implementation

One of the major objectives in the design of LoCo was to ensure finiteness of the logical models without forcing the knowledge engineer to finitely bound everything herself. This finiteness of configurations also gives us access to state-of-the-art software for solving combinatorial search problems via SAT solvers or constraint and integer programming. As already mentioned the idea of LoCo is to serve as a high-level source language that gets translated into a range of different executable target languages. To show that the formalism works in practice we have prototypically implemented a transformation to ASP, more precisely using the POTASSCO framework [Gebser et al., 2011] as a target language. The main motivation for this choice was twofold:

1. Being a dialect of rule-based logic programming [Gelfond, 2008], ASP allows a quite natural representation of LoCo problems, especially when extended with so-called cardinality constraints.

2. The POTASSCO framework [Gebser et al., 2011] gives us access to a state-of-the art conflict-driven clause learning solver much like contemporary SAT solvers. This technology has proven to be highly efficient and robust on numerous challenging academic and industrial problems.

While there are other mature systems implementing the answer set semantics like e.g. DLV [Leone et al., 2006], we also chose POTASSCO over those alternatives because it

proved to have the best performance at recent ASP competitions like e.g. [ASP Competition, 2011] and because of its active research community. In addition to ASP we have also developed a transformation to MiniZinc [Nethercote et al., 2007], providing us with access to a whole portfolio of integer programming, Sat and constraint solvers. However, a prototypical implementation for MiniZinc in the same way as for ASP has not been in the scope of this project and so the main ideas of this transformation will only be discussed on a theoretical level in section 5.3.

Concerning future work it would also be promising to develop an implementation that follows the idea of incrementally adding components instead of pre-generating all potentially useful ones. The idea to start from lower bound many components only to incrementally add components until a configuration is found is practically very appealing and would resemble the basic approach of generative CSP with additional upper bounds. However, there are currently no freely available solvers for generative or conditional Csp which we could use as a potential LoCo target language. Also the existing tool support concerning ASP does not yet meet our needs for implementing this idea [Gebser et al., 2008].

5.1 Workflow

Figure 5.1 depicts an overview of the transformation workflow. First the problem gets specified in the LoCo input language; domain and instance knowledge can optionally be defined together or separately in order to reuse a one-time definition of a problem specification with varying problem instances. The current version of the prototype only supports the representation of the LoCo knowledge via text-based input files. While this is sufficient for the purpose of a prototypical implementation, one of the main goals concerning future extensions is to develop a graphical user interface that creates the textual LoCo representation automatically. Next we parse the input files including a syntax and semantic check for correctness. The system is able to give the user appropriate and practical feedback in case of any errors. After that we determine the bounds for the generated components and check if the model is finite by traversing the configuration graph constructed during the parsing step. The specification is then translated into

executable code, resulting in one or more output files depending on the chosen target language and the chosen output mode. For example, the ASP implementation allows to translate instance and domain knowledge in separate files or everything altogether in one output file.

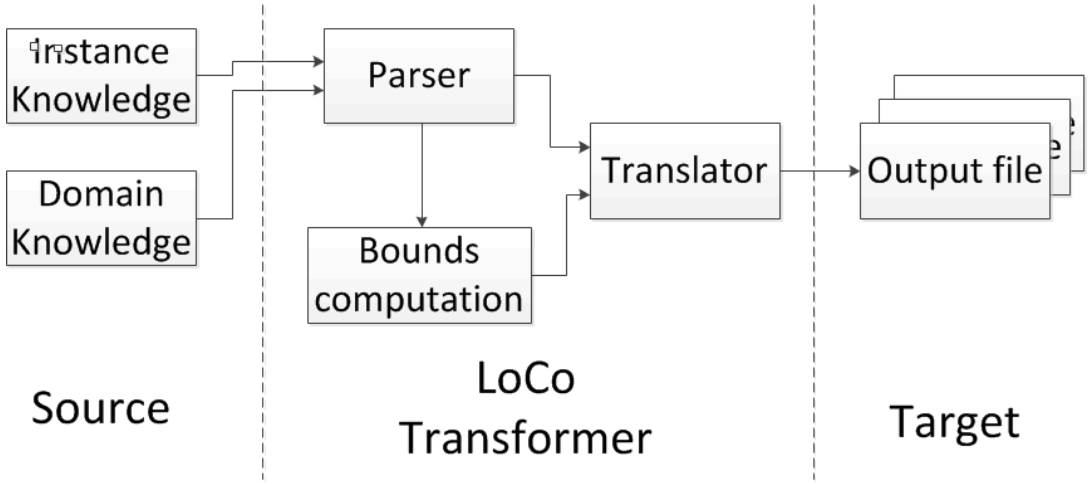


Figure 5.1: LoCo Workflow

The prototype has been implemented in *Java*. For parsing and analyzing the LoCo input language we use the *JavaCC* parser generator [JavaCC, 2014]. The bounds computation is implemented in ECL^{i}PS^e-Prolog [ECL^{i}PS^e, 2014] using the integer programming solver of the open-source optimization suite COIN-OR [COIN-OR, 2014]. Although the technical implementation has been a major part of the project in terms of both time and effort we decided not to include a detailed discussion of it since it would mainly involve a presentation about basic software engineering topics.

5.2 Transformation to ASP

The main focus of this section lies on discussing and presenting the automatically generated ASP code starting from a LIL specification. While we mention and explain the used ASP language constructs in a short and compressed way, a basic background knowledge

about logic programming and ASP in particular might be helpful for a better understanding. A detailed coverage of all needed topics though definitely goes beyond the scope of this thesis and instead we refer the reader to [Gebser et al., 2012, Brewka et al., 2011] for a deeper look into both ASP and Potassco.

Note that we explicitly represent all generated components that might be used in the configuration given by the derived finite upper bound. During the solving process LoCo then picks a subset of these generated components, possibly a minimal subset for an optimization problem. In case the specification can be satisfied and a solution is found, the chosen components form a valid model.

5.2.1 Constant and Component definitions

Constants We first depict the manually defined constants of our model in ASP code in listing 5.1. The transformation output is rather self-explanatory since it is highly similar and more or less a one-to-one mapping from the LoCo input language definition in listing 4.1 of the previous chapter.

```
1  #const hMaxHeight = 10.
2  #const hMaxWidth = 5.
3  #const maxSizeTA = 10.
4  #const maxSizeTB = 12.
5  #const cMaxNrTA = 5.
6  #const cMaxNrTB = 3.
7  #const cMaxSizeTA = 5 * maxSizeTA.
8  #const cMaxSizeTB = 7 * maxSizeTB.
9  #const cMaxSize = cMaxSizeTA + cMaxSizeTB.
10 #const rMaxPos = 4.
```

Listing 5.1: Definition of user-defined constants in ASP

In addition to these manual constants we need to create constants representing the upper bounds for the input components (listing 5.2). This information can be defined on either level or can even be separated between the two as it has been outlined in the previous chapter. In case we define a bound for a component on the instance level that

already received a value on the domain level, the instance level value will have priority in this situation and will be the one used at this point. Since the internal component IDs simply run starting from 1 in ascending order up to their defined upper bounds, there is no need to state lower bounds for input components and the upper bounds at the same time represent the number of given instances.

```
1 % Upper bound for input component person
2 #const person_UB = 2.
3 % Upper bound for input component house
4 #const house_UB = 1.
5 % Upper bound for input component thingA
6 #const thingA_UB = 5.
7 % Upper bound for input component thingB
8 #const thingB_UB = 5.
```

Listing 5.2: Upper bounds for input components

In contrast to this generated components come with both a lower and an upper bound as shown in listing 5.3. Of course the upper bound is the important one here to maintain the finiteness of the model and in case of an undefined lower bound we could assume it to be one without any real ill effects. The automatic bounds computation nevertheless also computes a lower bound for every generated component which helps to reduce the domain sizes and consequently the total search space for the problem. We investigate in more detail how these bounds constants are practically used further below.

Input component definition Listing 5.4 depicts the component definition for input components house and thingA. As already discussed the ASP transformation creates all possible component instances from which it then picks a subset during the solving process. Hence our knowledge base contains upper bound many instances for all component types. For this purpose the predicates named after the component with an added "Gen" as a postfix are determined to fulfill this role, i.e. predicate houseGen in line 2 representing the finitely many component instances which *might* be used in the configuration for component house. The instances of predicate house on the other hand are those that

```
1 % Setting computed lower bound for generated component cabinet
2 #const cabinet_LB = 2.
3 % Setting computed upper bound for generated component cabinet
4 #const cabinet_UB = 10.
5 % Setting computed lower bound for generated component room
6 #const room_LB = 1.
7 % Setting computed upper bound for generated component room
8 #const room_UB = 10.
```

Listing 5.3: Bounds for generated components

actually *are* used as part of a solution set. Since in the case of input components the number of effective instances is equal to the upper bound, all of these generated instances will be actually used. So every houseGen object automatically becomes a house object (line 3). This is the major difference in comparison to the ASP representation of generated components as we shall see below.

Lines 4 and 5 represent the two defined house attributes height and width. In order to maintain uniqueness of predicate names the component name is put as a prefix to every attribute name, e.g. attribute height becomes houseHeight in the ASP code. Both are Integer attributes with bounds of [*0..hMaxHeight-1*] and [*0..hMaxWidth-1*] respectively.

Let's have a quick look at the output code for these attributes: Both lines use so-called *cardinality constraints* [Simons et al., 2002] which are a special case of weighted constraint rules and a form of aggregate constraints for counting the number of occurrences. The numbers outside the curly brackets represent the cardinality limits and for the purpose of attribute definitions both the upper and lower cardinalities are set to 1 since every house instance should have exactly one width attribute. Inside this choice construct there is a conditional construct selecting only those literals in front of the colon for which the conditional part afterwards holds. Lines 5 expresses the fact that for every `H` serving as an identifier of a house instance there is exactly one ground instance of the predicate `houseWidth(H,HW)` whereas `HW` represents the width attribute. The conditional part [*HW=0..hMaxWidth-1*] represents the attribute bounds and restricts the attribute value to be between zero and [*hMaxWidth-1*], with hMaxWidth being a manually-defined

```
1  % Input component house
2  houseGen(1..house_UB).
3  house(H) :- houseGen(H).
4  1 { houseHeight(H, HH) : HH = 0..hMaxHeight-1 } 1 :- house(H).
5  1 { houseWidth(H, HW) : HW = 0..hMaxWidth-1 } 1 :- house(H).
6
7  % Input component thingA
8  thingAGen(1..thingA_UB).
9  thingA(T) :- thingAGen(T).
10 1 { thingASize(T,TS) : TS = 0..tMaxSizeA } 1 :- thingA(T).
11 1 { thingABig(T,TB) : TB = 0..1 } 1 :- thingA(T).
12 1 { thingADirty(T,TD) : TD = 0..1 } 1 :- thingA(T).
13 1 { thingAForUpper(T,TF) : TF = 0..1 } 1 :- thingA(T).
14 1 { thingAMinFloor(T,TM) : TM = 0..hMaxHeight-1 } 1 :- thingA(T).
15 1 { thingAMaxFloor(T,TMA) : TMA = 0..hMaxHeight-1 } 1 :- thingA(T).
```

Listing 5.4: Input components house and thingA

constant from above. For a deeper look into this and other related ASP and Potassco language elements we again refer to [Gebser et al., 2012, Simons et al., 2002].

In an analogous manner to a house component we illustrate the full encoding for a thingA component in lines 8-15. Next to some Integer attributes this component definition contains attributes big, dirty and forUpper of type Boolean in lines 11-13. Internally Boolean attributes are mapped to the interval [0,1] as can be seen in the conditional parts of the cardinality constructs.

Generated component definition The definition for a generated component is highly similar to the one for input components we've just covered. In fact, the representation of attributes is identical and the only difference concerns the selection and generation process of instances. Remember that for every generated component we select a subset of the possible instances with its cardinality between a lower and an upper bound instead of making every cabinetGen object automatically a cabinet object. The selection process is done via a *choice rule* [Niemelä et al., 1999] in line 3 which is basically another variant of using the weighted constraint syntax of a cardinality constraint in Potassco. Inside the

```
 1 % Generated component cabinet
 2 cabinetGen(1..cabinet_UB).
 3 cabinet_LB { cabinet(C) : cabinetGen(C) }.
 4 cabinet(C) :- cabinetGen(C), cabinetGen(C2), cabinet(C2), C < C2.
 5 1 { cabinetSize(C,CS) : CS = 0..cMaxNrTA*tMaxSizeA+cMaxNrTB*tMaxSizeB } 1
 6   :- cabinet(C).
 7 1 { cabinetDirty(C,CD) : CD = 0..1 } 1 :- cabinet(C).
 8 1 { cabinetBig(C,CB) : CB = 0..1 } 1 :- cabinet(C).
 9 1 { cabinetTop(C,CT) : CT = 0..1 } 1 :- cabinet(C).
10 1 { cabinetColor(C,CC) : cabinetColorEnum(CC) } 1 :- cabinet(C).
11 cabinetColorEnum(red).
12 cabinetColorEnum(blue).
13 cabinetColorEnum(green).
```

Listing 5.5: Generated component cabinet

curly brackets the conditional construct [*cabinet(C) : cabinetGen(C)*] is a short-cut for a collection of all possible cabinet instances C instantiated with variables from cabinetGen. The lower bound on the left side states that from the available collection of instances there have to be at least lower bound many selected. On the right side of the curly brackets we can omit the upper bound since the maximum number of occurrences is already implicitly bounded by the domain of cabinetGen in line 2. Line 4 represents a symmetry breaking constraint for the selection of cabinets s.t. instances with lower ID get priority. Note that this additional constraint is not necessary for input components since we don't make a selection for them. Complex expressions for attribute bounds are easy to map since the conditional part of a conditional construct in Potassco can also contain arithmetic expressions (line 5). The subsequent lines show some attributes of the same structure as discussed previously. Attribute color in line 10 is again more interesting since it is an enumeration type. Instead of a specified range in the conditional part we use a conjunction of domain predicates for attributes of this type. The available options are defined as simple facts, e.g. colors *red*, *blue* and *green* in lines 11-13.

5.2.2 Connection rules

We now cover the transformation of rules and start with the supported forms of connection rules. In order to maintain a uniform persistent structure we will map the same snapshots of the House Problem that were already discussed in chapters 3 and 4. Since the mappings of some ϕ constraint parts get very large, most of them for both the connection and consistency rules will be presented in a shortened way or left out for the sake of better clarity and presentation. We will discuss the main constructs for representing a constraint on the basis of a standard binary connection and after that only discuss additional elements that haven't been presented before or which are specific to the type of mapped rule.

Standard binary connections

The code snippet in listing 5.6 shows the transformation of a standard binary connection between thingA and cabinet from the axiomatization of example 2 in chapter 3. The first line represents the existence enforcement of the required binary connection predicate by again using cardinality constraints [Simons et al., 2002] with the numbers or constants outside the curly brackets specifying the lower and upper bounds: In more detail it expresses the fact that there is exactly one ground instance of the predicate `thingA_Cab(TA,C)` for every `TA` such that `C` and `TA` are identifiers of cabinets and things of type A. Conditional literals are used for the generation of the binary connection predicates consisting of a main part and a conditional part, e.g. `thingA_Cab(T,C):cabGen(C)` in line 1. From this it follows that every id `C` used in the generated connection predicate must belong to a valid cabinet instance. The conditional part (`cabGen(C)` in this case) for such rules must be specified by ground facts in the knowledge base as done previously during the definition of bounds for the component types. Hence our knowledge base contains all possible instances of `cabGen`, i.e. the finitely many component instances which *might* be used in the configuration. The instances of `cab` and `thingA` are those that actually *are* used as part of a solution set. The integrity constraint in line 3 then ensures that every cabinet that features in a connection is also in the extension of the `cab` predicate.

```
1  % Binary connection thingA -> cabinet
2  1 { thingA_cabinet(TA,C) : cabinetGen(C) } 1 :- thingA(TA).
3  :- thingA_cabinet(TA,C), not cabinet(C).
4  % Phi Constraint
5  :- not c1(TA,C), thingA_cabinet(TA,C).
6  c1(TA,C) :- c1_1(TA,C).
7  c1(TA,C) :- c1_2(TA,C).
8  c1_1(TA,C) :- c1_1_1(TA,C), c1_1_2(TA,C).
9  c1_1_1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), cabinetBig(C,CB), CB == 1.
10 c1_1_2(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingABig(TA,TAB), TAB == 1.
11 c1_2(TA,C) :- not c1_2_1(TA,C), thingA_cabinet(TA,C), thingA(TA), cabinet(C).
12 c1_2_1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingABig(TA,TAB), TAB == 1.
```

Listing 5.6: ASP output for the connection from thingA to cabinet

Lines 5-12 depict the mapping of the constraint subformula from example 2. The integrity constraint in line 5 states that for every connection between a thing and a cabinet the constraint `c1` must hold. Every ϕ constraint of a connection or consistency rule gets assigned a distinct number s.t. the name of the corresponding constraint predicate is `c` concatenated with this allocated number. The following lines 6-7 represent the mapping of a disjunction where either one of them has to hold in order to justify `c1`.

When mapping a logical connective we add nested constraint predicates with new names created by further concatenating an underscore with another running number in ascending order. In the current case the left- and right-hand side expressions of the disjunction are represented by `c1_1` and `c1_2`. The left disjunction is separated furthermore in two conjunction operand expressions (line 8). Lines 9 and 10 then check for both of these parts if attribute big of cabinet and thingA respectively is set to TRUE. The second part of the disjunction on the top level contains a negation connective, represented here in that the body of line 11 is fulfilled when the rule in line 12 fails.

In the just described way the generated connection predicates of a ϕ constraint form a kind of expression tree where the fulfillment of a predicate depends on the fulfillment of lower-level predicates or on an expression in case it is a leaf predicate. Figure 5.2 shows the structure for the predicates we have just described in the previous paragraph for a better understanding.

Next to transforming ϕ constraints in the presented way the translator component of

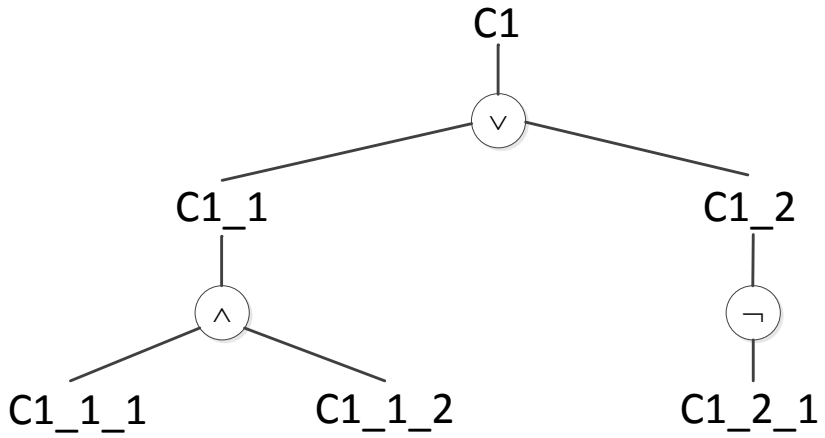


Figure 5.2: Predicate tree for listing 5.6

the prototypical implementation also fulfills some additional optimization steps on top of that for simplifying their structure. The main purpose here is to decrease the number of needed predicates as well as to reduce the depth of the resulting predicate tree.

Listing 5.7 depicts the reduced version of the ϕ constraint. We present the main undertaken steps from the bottom up: First we can eliminate the negation from lines 11-12 in the original constraint since it is an attribute comparison and we can simply change it to comparing attribute big with zero instead of one. The optimizer also eliminates the conjunction in lines 8-10 and reduces it to one ASP rule (lines 3-4 in the new constraint). Since we could remove all logical connectives of the tree on the lower levels the disjunction can also be moved up one level by eliminating the helper predicates `c1_1` and `c1_2`.

```
1  % Simplified Phi Constraint
2  :- not c1(TA,C), thingA_cabinet(TA,C).
3  c1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingBig(TA,TAB), cabinetBig(C,CB),
4      TAB == 1, CB == 1.
5  c1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingABig(TA,TAB), TAB == 0.
```

Listing 5.7: Modified ϕ constraint

Side note: The attentive reader might have noticed that there are other different and easier ways to formulate the original constraint. In fact since the LoCo input language

and the translator component also support logical implication, the constraint `C.big -> TA.big` would be logically equivalent.

Listing 5.8 shows the reverse direction of the binary connection from cabinet to thingA. For this and all remaining connection and consistency rules we will only present the already optimized constraint parts. While the basic structure is analogous to the rule in the other direction, the transformation of a summation aggregate in lines 7-10 is the interesting topic that shall be discussed here. The main constraint part is shown in lines 9-10, ensuring that the calculated sum `TAS_SUM` is at most 5. `TAS_SUM` represents the sum of all thingA sizes for a certain cabinet `C` in extension of predicate `c2_Aggr1` (line 7) which represents the combination of the connection predicate with the size attribute of thing. However, `c2_Aggr1` only contains those tuples that satisfy the nested constraint `c2_Constr1` (line 8) ensuring that only big things will be taken into consideration. The structure of a nested constraint is analogous to a standard constraint. For a more thorough and deeper understanding of the mentioned ASP language elements here we refer again to [Gebser et al., 2012, Brewka et al., 2011].

```
1  % Binary connection cabinet -> thingA
2  0 { thingA_cabinet(TA,C) : thingAGen(TA) } 5 :- cabinet(C).
3  :- thingA_cabinet(TA,C), not thingA(TA).
4  % Phi Constraint
5  :- not c2(TA,C), thingA_cabinet(TA,C).
6  c2(TA,C) :- c2_1(TA,C), c2_2(TA,C).
7  c2_1_Aggr1(C,TA,TAS) :- thingA_cabinet(TA,C), thingASize(TA,TAS), c2_1_Constr1(TA,C).
8  c2_1_Constr1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingABig(TA,TAB), TAB == 1.
9  c2_1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C),
10     TAS_SUM = #sum [ c12_1_Aggr1(C,_,TAS) = TAS], TAS_SUM <= cMaxSizeTA.
11 c2_2(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), cabinetTop(C,CT), CT == 0.
12 c2_2(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingAForUpper(TA,TAF), TAF == 1.
```

Listing 5.8: ASP output for the binary connection from cabinet to thingA

Unfolded Binary Connections

Let's now have a look on the transformation of unfolded binary connections on the basis of the LIL rule 4.5 from chapter 4. At this point we will only discuss the additional elements

which are necessary for this type of axiom. The choice rule in line 1 stipulates that for every position P either one of the two cases positionUBC1(P) or positionUBC2(P) has to hold. Every possible case then determines the number of connected cabinets for it (lines 2-3). The constraint for the first case in lines 5-6 has the same structure as the constraints for the standard binary connections discussed previously.

Case 2 is more interesting since we introduce the addressing of specific component instances. The integrity constraint in line 8 enforces the existence of a fact positionUBC2Constr(P,C) for every case 2. In line 9 we introduce additional variables C1 and C2 for representing the component instances. This way positionUBC2Constr(P,C) must hold for every position P connected to a cabinet C and for the fulfillment of constraint predicate c4 the two specific component instances C1 and C2 are added and need to satisfy the conditions expressed in the constraint part of case 2. Thus, by using this construct C1 and C2 can be seen as existentially quantified variables.

The body of the lower-level predicates c4_1 and c4_2 contains cabinet atoms for the variables C, C1 and C2. An attribute addressed via C, e.g. attribute big in c4_1, refers to all cabinet instances connected to a position s.t. none of the cabinet instances are allowed to be big. On the other hand the top attribute addressed by C1 and C2 respectively refers in each case to only one specific instance whereupon the instances referenced by C1 and C2 have to be different from each other.

One-to-many connections

Listing 5.10 shows the ASP output for the standard one-to-many connection from example 4.8. We chose to present the transformation of this instead of example 4.9 because the constraint part contains counting aggregates which haven't been discussed as of yet. Similar to binary connections we employ a choice rule in line 1 to enforce the required connections for every cabinet instance. In this case though the cardinality construct in the head contains two conditional literals each for connections with thingA and thingB respectively to choose from. The lower cardinality is set to one and the upper cardinality is undefined, expressing the fact that for each cabinet C at least one connection atom thingA_cabinet(TA,C) or thingB_cabinet(TB,C) needs to exist.

```
 1  1 { positionUBC1(P), positionUBC2(P) } 1 :- position(P).
 2  1 { cabinet_position(C,P) : cabinetGen(C) } 1 :- positionUBC1(P).
 3  2 { cabinet_position(C,P) : cabinetGen(C) } 2 :- positionUBC2(P).
 4  % Case 1
 5  :- not c3(P,C), positionUBC1(P), cabinet_position(C,P), cabinet(C), position(P).
 6  c3(P,C) :- positionUBC1(P),cabinet_position(C,P),cabinet(C),position(P),cabinetTop(C,CT),CT == 0.
 7  % Case 2
 8  :- not positionUBC2Constr(P,C), positionUBC2(P), cabinet_position(C,P), cabinet(C), position(P).
 9  positionUBC2Constr(P,C) :- c4(P,C,C1,C2).
10  c4(P,C,C1,C2) :- c4_1(P,C,C1,C2), c4_2(P,C,C1,C2).
11  c4_1(P,C,C1,C2) :- positionUBC2(P), cabinet_position(C,P), cabinet(C), position(P),
12    cabinet_position(C1,P), cabinet(C1), cabinet_position(C2,P), cabinet(C2), C1!=C2,
13    cabinetBig(C,CB), CB == 0.
14  c4_2(P,C,C1,C2) :- positionUBC2(P), cabinet_position(C,P), cabinet(C), position(P),
15    cabinet_position(C1,P), cabinet(C1), cabinet_position(C2,P), cabinet(C2), C1!=C2,
16    cabinetTop(C1,C1T), cabinetTop(C2,C2T), C1T == 1, C2T == 0.
17  c4_2(P,C,C1,C2) :- positionUBC2(P), cabinet_position(C,P), cabinet(C), position(P),
18    cabinet_position(C1,P), cabinet(C1), cabinet_position(C2,P), cabinet(C2), C1!=C2,
19    cabinetTop(C1,C1T), cabinetTop(C2,C2T), C1T == 0, C2T == 1.
```

Listing 5.9: ASP output for an unfolded binary connection

Let's have a look at the ϕ constraint and in particular on the transformation of counting aggregates: Similar to the summation aggregate in listing 5.8 each of the counting aggregates has a nested constraint for restricting the counting of instances to only big things of type A or B respectively. The structure for these nested constraints is identical to the summation aggregate; see lines 5-6 for thingA and lines 7-8 for thingB. In line 10 we use the built-in counting aggregates of Potassco to assign the number of counted instances to variables `TA_CNT` and `TB_CNT` and state that their sum needs to be smaller or equal to 4 (`TA_CNT + TB_CNT <= 4`). Note that in case there are only connections from a cabinet to either thingA or to thingB then the counting aggregate with no elements simply returns zero.

Exclusive-OR One-to-Many connections

Following the same pattern as for the other connection types discussed so far we depict the ASP code for the LIL example 4.10 from chapter 4. Since in this case cabinets can

```
1  % One-to-many connection
2  1 { thingA_cabinet(TA,C) : thingAGen(TA), thingB_cabinet(TB,C) : thingBGen(TB) }  :- cabinet(C).
3  % Phi Constraint
4  :- not c5(C,TA,TB),cabinet(C),thingA(TA),thingB(TB),thingA_cabinet(TA,C),thingB_cabinet(TB,C).
5  c5_Aggr1(C,TA) :- thingA_cabinet(TA,C), c5_AggrConstr1(TA,C).
6  c5_AggrConstr1(TA,C) :- thingA_cabinet(TA,C), thingA(TA), cabinet(C), thingABig(TA,TAB), TAB==1.
7  c5_Aggr2(C,TB) :- thingB_cabinet(TB,C), c5_AggrConstr2(TB,C).
8  c5_AggrConstr2(TB,C) :- thingB_cabinet(TB,C), thingB(TB), cabinet(C), thingBBig(TB,TBB), TBB==1.
9  c5(C,TA,TB) :- cabinet(C), thingA(TA), thingB(TB), thingA_cabinet(TA,C), thingB_cabinet(TB,C),
10     TA_CNT = #count{c5_Aggr1(C,_)}, TB_CNT = #count{c5_Aggr2(C,_)}, TA_CNT + TB_CNT <= 4.
```

Listing 5.10: ASP output for a standard one-to-many connection

be only connected to either things of type A or to type B, the transformation starts with a choice rule requiring to select either case `cabinetXORC1(C)` or case `cabinetXORC2(C)` (line 2). Lines 3-4 contain the ASP code for the first case: We stipulate in line 3 that for every cabinet C and case one there need to exist between 1 and 5 connections to thingA. Line 4 expresses the fact that the existence of at least one connection with thingA enforces case one to hold for this specific cabinet. This reverse direction is needed to guarantee the exclusivity of the two cases. Lines 5 and 6 represent the analogous mapping for thingB. As already mentioned, XOR connections don't possess ϕ constraints of any kind.

```
1  % XOR One-to-many connection
2  1 { cabinetXORC1(C), cabinetXORC2(C) } 1 :- cabinet(C).
3  1 { thingA_cabinet(TA,C):thingAGen(TA) } 5  :- cabinetXORC1(C).
4  cabinetXORC1(C) :- thingA_cabinet(_,C).
5  1 { thingB_cabinet(TB,C):thingBGen(TB) } 3  :- cabinetXORC2(C).
6  cabinetXORC2(C) :- thingB_cabinet(_,C).
```

Listing 5.11: ASP output for a XOR one-to-many connection

General One-to-Many connections

The last type of one-to-many connection is the most expressive form which provides the opportunity to address individual component instances. Same as before the transformation starts out with a separation into distinct cases (line 2); we show here the 4 cases from rule 4.11. The exact number of involved instances for each component type is also

defined by choice rules, e.g. case one in lines 4-5 stipulates exactly one connection to each type of thing. Lines 13-14 show that next to an exact number of instances a case can also cover an interval of allowed instances; case 4 here subsumes every configuration with 4 to 7 connected thingA instances and 5 to 7 thingB instances.

Every case can also have an optional constraint part. We depict the constraint for case 2 in lines 16-19 which stipulates that the size of at least one thingA instance needs to be bigger than all connected thingB instances. The structure of the constraint and the addressing of specific instances is highly similar to the example that has been discussed for unfolded binary connections.

```
1  % General one-to-many connection
2  1 { cabinetGI2MC1(C), cabinetGI2MC2(C), cabinetGI2MC3(C), cabinetGI2MC4(C) } 1 :- cabinet(C).
3  % Case 1
4  1 { cabinet_thingA(C,TA) : thingAGen(TA) } 1 :- cabinetGI2MC1(C).
5  1 { cabinet_thingB(C,TB) : thingBGen(TB) } 1 :- cabinetGI2MC1(C).
6  % Case 2
7  2 { cabinet_thingA(C,TA) : thingAGen(TA) } 2 :- cabinetGI2MC2(C).
8  3 { cabinet_thingB(C,TB) : thingBGen(TB) } 3 :- cabinetGI2MC2(C).
9  % Case 3
10 3 { cabinet_thingA(C,TA) : thingAGen(TA) } 3 :- cabinetGI2MC3(C).
11 4 { cabinet_thingB(C,TB) : thingBGen(TB) } 4 :- cabinetGI2MC3(C).
12 % Case 4
13 4 { cabinet_thingA(C,TA) : thingAGen(TA) } 7 :- cabinetGI2MC4(C).
14 5 { cabinet_thingB(C,TB) : thingBGen(TB) } 7 :- cabinetGI2MC4(C).
15 % Phi constraint for Case 2
16 :- not cabinetGI2MC2Constr(C,TB), cabinetGI2MC2(C), cabinet(C), cabinet_thingB(C,TB), thingB(TB).
17 cabinetGI2MC2Constr(C,TB) :- c6(C,TB,TA1).
18 c6(C,TB,TA1) :- cabinetGI2MC2(C), cabinet(C), cabinet_thingB(C,TB), thingB(TB),
19     cabinet_thingA(C,TA1), thingA(TA1), thingASize(TA1,TA1S), thingBSize(TB,TBS), TA1S > TBS.
```

Listing 5.12: ASP output for a general one-to-many connection

5.2.3 Consistency rules

Let's now investigate the transformation of the non-local constraint parts of consistency rules.

Candidate key rules

ASP is well-suited to express candidate key axioms in a convenient and natural way. We look at the transformation of both examples from chapter 4, starting with rule 4.12. The set of attributes which uniquely identifies a component instance is represented in standard constraint form. Constraint c7 checks the equivalence of attributes floor (line 3) and pos (line 4). If constraint c7 holds for two room instances R1 and R2 then the integrity constraint in line 2 only holds if expression `R1 != R2` fails, i.e. both instances have to be equal. This condition represents the consequent part of the axiom.

```
1 % Candidate key rule
2 :- R1 != R2, room(R1), room(R2), c7(R1,R2).
3 c7(R1,R2) :- room(R1), room(R2), roomFloor(R1,R1F), roomFloor(R2,R2F), R1F == R2F,
4     roomPos(R1,R1P), roomPos(R2,R2P), R1P == R2P.
```

Listing 5.13: ASP output for a candidate key rule

Listing 5.14 then shows the extended version of a cardinality key axiom by taking also connections to other component types into account. Next to having the identical nr attribute value, two positions have to be equal in case they belong to the same room instance. One could certainly see the incorporation of connection predicates as just a comparison of foreign keys and therefore not vastly different from standard component attributes.

Connection-generating rules

We depict the transformation of the connection-generating rule from example 10 in listing 5.15. The body of the ASP rule represents the antecedent and contains the chain of connected components. In this case the mapping can be reduced to a single ASP rule.

```
1  % Candidate key rule
2  :- P1 != P2, position(P1), position(P2), room(R),
3      position_room(P1,R), position_room(P2,R), c8(P1,P2,R).
4  c8(P1,P2,R) :- position(P1), position(P2), room(R), position_room(P1,R), position_room(P2,R),
5      positionNr(P1,P1N), positionNr(P2,P2N), P1N == P2N.
```

Listing 5.14: ASP output for an extended candidate key rule

If the antecedent would also consist of additional attribute comparisons or arithmetic expressions, it would be modelled as a separate constraint in the usual way with the body of the main rule being replaced by the respective constraint predicate.

```
1  % Connection-generating rule for connection room_person(RO,PERS)
2  room_person(RO,PERS) :- person(PERS), thingB(TB), cabinet(CAB), position(POS), room(RO),
3      person_thingB(PERS,TB), thingB_cabinet(TB,CAB),
4      cabinet_position(CAB,POS), position_room(POS,RO).
```

Listing 5.15: ASP output for a connection-generating rule

General First Order rules

GFO rules are able to express more complex non-local constraints. They are also used for modelling intensional component catalogue knowledge. Following the same pattern as for the other connection and consistency rules we show the ASP mapping for the respective examples that have been introduced in axiom form in chapter 3 and in LIL form in chapter 4.

We start with example 11 for coloring cabinets blue in case they are placed on positions 1 or 2 : The transformation is broken down into 2 constraint parts with the constraint identified by predicate `c9` representing the antecedent and predicate `c10` representing the consequent of the constraint. The integrity constraint in line 2 then determines that `c10` has to hold for a given cabinet-position connection if pre-condition `c9` is fulfilled.

The second GFO example in listing 5.17 shows an application of this rule type for a slightly extended connection-generating rule where the consequent doesn't solely consist of a connection that needs to exist but of a potentially more complex ϕ constraint part (in

```
1 % general FO rule
2 :- not c10(C,P), cabinet(C), position(P), cabinet_position(C,P), c9(C,P).
3 c9(C,P) :- cabinet(C), position(P), cabinet_position(C,P), positionNr(P,PN), PN == 1.
4 c9(C,P) :- cabinet(C), position(P), cabinet_position(C,P), positionNr(P,PN), PN == 2.
5 c10(C,P) :- cabinet(C), position(P), cabinet_position(C,P), cabinetColor(C,CC), CC == blue.
```

Listing 5.16: ASP output for a general FO rule (1)

this case some attribute comparisons). The connection chain in the integrity constraint enforces the fulfillment of constraint predicate `c11` which is then simply modelled in standard constraint form.

```
1 % general FO rule
2 :- not c11(TB,R,C,P), thingB(TB), room(R), cabinet(C), position(P), thingB_cabinet(TB,C),
3     cabinet_position(C,P), position_room(P,R).
4 c11(TB,R,C,P) :- thingB(TB), room(R), cabinet(C), position(P), thingB_cabinet(TB,C),
5     cabinet_position(C,P), position_room(P,R), thingBMinFloor(TB,TBM), roomFloor(R,RF),
6     TBM <= RF, thingBMaxFloor(TB,TBMA), roomFloor(R,RF), TBMA >= RF.
```

Listing 5.17: ASP output for a general FO rule (2)

The final GFO example in listing 5.18 covers the introduction of additional variables in the consequent part of the rule. Like for the first GFO mapping in listing 5.16 there exists a pre-condition represented here by constraint part `c12(TA,C)`. Line 4 is key for the mapping of the existential quantifier and expresses the fact that `c14(TA,C)` holds when `c13(TA,C,P)` is fulfilled. Variable `P` here is the newly introduced variable representing an arbitrary person component instance. Hence the mapping for the introduction of a new variable via an existential quantifier is analog to the way we address component instances in unfolded binary connections (see also the mapping from listing 5.9) .

5.2.4 Domain and Instance Knowledge

Having now finished the presentation of connection and consistency rules, let's shift our attention to the mapping of configuration problem specifications.

```
1  :- not c14(TA,C), thingA(TA), cabinet(C), thingA_cabinet(TA,C), c12(TA,C).
2  c12(TA,C) :- thingA(TA), cabinet(C), thingA_cabinet(TA,C),
3      thingADirty(TA,TAD), cabinetDirty(C,CD), TAD == CD.
4  c14(TA,C) :- c13(TA,C,P).
5  c13(TA,C,P) :- thingA(TA), cabinet(C), person(P), person_thingA(P,TA).
```

Listing 5.18: ASP output for a general FO rule (3)

Domain knowledge We can tick off the transformation of ports since it is identical to connection axioms as well as the transformation of attribute ranges since this has already been discussed as part of the component definitions in section 5.2.1. The mapping for the intensional Component Catalogue rules is identical to the mapping of General First Order rules, so that concerning the Domain Knowledge this leaves the mapping of the extensional Component Catalogue as the only open topic that needs to be addressed specifically at this point.

Extensional Component Catalogue Listing 5.19 depicts the ASP code for the extensional catalogue from example 15 of chapter 3: For every instance of thingA exactly one of the possible attribute tuples needs to be selected (line 1). The selected tuple option then assigns values to all component attribute values, e.g. tuple 1 implies the attribute size being set to 5 (line 3), the Boolean attribute big being set to TRUE (line 4) and continues to assign values in the same manner to all remaining attributes of thingA. In order to create a dependency in both directions the existence of the complete attribute vector also implies the selection of tuple 1 (lines 7-8), i.e. the attribute vector uniquely identifies the selected tuple option and vice versa. The mapping of further tuples is done the same way as just described for the first one.

Instance knowledge Same as for the Domain Knowledge the main parts of the ASP transformation for the Instance Knowledge have already been described. While the instance knowledge can define additional constants, their mapping at this point is identical to the standard constants in listing 5.1. The parser component detects if constants get a new value at this level s.t. in case of multiple values there will be only one assignment for each constant in the ASP output code. The same process also applies when setting

```
1  1 { thingATuple1(TA), thingATuple2(TA), ... } 1 :- thingA(TA).
2  % Tuple 1
3  thingASize(TA,5) :- thingATuple1(TA).
4  thingABig(TA,1) :- thingATuple1(TA).
5  thingADirty(TA,0) :- thingATuple1(TA).
6  ...
7  thingATuple1(TA) :- thingA(TA), thingASize(TA,5), thingABig(TA,1), thingADirty(TA,0),
8      thingAForUpper(TA,0), thingAMinFloor(TA,0), thingAMaxFloor(TA,2).
9  % Tuple 2
10 thingASize(TA,3) :- thingATuple2(TA).
11 ...
```

Listing 5.19: Extensional component catalogue for thingA

manual bounds resulting in multiple values for bounds constants. The ASP mapping of bounds for both input and generated components has been presented before in listings 5.2 and 5.3.

Let's now investigate the mapping of component knowledge, starting with the input components. The according ASP code for the input component knowledge of listing 4.5 from chapter 4 is depicted in listing 5.20:

```
1  #const person_UB = 2.
2
3  #const house_UB = 1.
4  houseHeight(1,5).
5  houseWidth(1,2).
6
7  #const thingA_UB = 5.
8  thingASize(1,3).
9  thingABig(1,1).
10 thingADirty(1,0).
11 ...
12 thingASize(2,4).
13 thingABig(2,1).
```

Listing 5.20: Input component knowledge

It is mandatory that for all input components the number of instances needs to be stated. Line 1 represents the mapping for the LIL specification `#person = 2`. Remember

that when defining component person the number of instances are defined via predicate `personGen(1..person_UB)`, so declaring a constant `person_UB` and setting its value to 2 gives the desired outcome. We get the same type of mapping for components house and thingA in lines 3 and 7. The remaining lines contain the corresponding attribute predicates, e.g. `houseHeight(1,5)` in line 4 states that attribute height of the house instance identified by ID 1 gets value 5. Values for Boolean attributes are represented by 1 and 0, s.t. `thingABig(2,1)` in line 13 sets attribute big of the thingA instance 2 to being TRUE.

The mapping of input connections to ASP in listing 5.21 is quite straightforward since the LIL representation of connection predicates is equal to the one in ASP (lines 5-9). In order to eliminate the possibility of invalid IDs the rules in lines 2-3 make sure that the IDs used in the connections belong to component instances of person or thingA respectively.

```
1 % Input connections for person_thingA
2 person(P) :- person_thingA(P,_).
3 thingA(T) :- person_thingA(_,T).
4
5 person_thingA(1,1).
6 person_thingA(1,2).
7 person_thingA(1,3).
8 person_thingA(1,4).
9 person_thingA(1,5).
```

Listing 5.21: Input connections between persons and things

The input component and connection knowledge can be seen as a mandatory part of a partial configuration that needs to be defined for every problem instance. Next to component and connection knowledge of type input for which it is imperative to be specified on the instance level we can optionally also specify component bounds, attribute values and the existence of specific component and connection instances for generated components. Listing 5.22 shows the transformation of some partial configuration knowledge from the LIL specification of listing 4.7.

```
1  #const cabinet_UB = 7.
2  #const cabinet_LB = 1.
3  room(5).
4  roomSize(5,18).
5  roomFloor(5,2).
6  roomRoomID(7,R005).
7  roomSize(7,15).
8  ...
9  cab_thingB(2,3).
```

Listing 5.22: Partial configuration knowledge

Since cabinet is a generated component the mapping for `#cabinet = 7` results in an ASP output of constants for both the upper and lower bound (lines 1-2). The automatic bounds computation potentially overwrites lower bounds, e.g. suppose it computes a lower bound of 3 then `cabinet_LB` in line 3 would be set to 3 instead. Ground facts like `room(5)` in line 3 have to the same form in both LIL and ASP and so don't need any type of processing. The next lines contain some attribute value assignments (lines 4-7) and an example of a single connection instance (line 9).

Listing 5.23 shows how to exclude component and connection instances from the model. The mapping to ASP again is rather simple by transforming negated facts into integrity constraints.

```
1  :- room(4).
2  :- cab_thingB(2,4).
```

Listing 5.23: Excluding components and connections

5.2.5 Remarks

Default vs. classical negation

An important point that needs to be addressed is that the source and target languages may have different semantics in terms of negation. In particular, the LoCo axiomatization uses classical (or strong) negation whereas ASP was primarily conceived using default

negation [Gelfond and Lifschitz, 1991] and only later extended to support classical negation too. Default negation basically expresses the fact that a literal not L holds by default unless L is derived whereas the classical negation of L holds only if the complement of the proposition, expressed by $-$L, can be derived.

Our approach for solving this is to use classical negation throughout the whole workflow. However, LoCo axiomatizations allow the negation of arbitrarily complex constructs, e.g. in the ϕ constraint subformulas of rules. This could involve an arithmetic expression or a combination of several expressions by logical connectives whereas classical negation in ASP can be applied to atoms only. We sidestep this issue by transforming every ϕ formula into negation normal form (NNF) such that negation occurs only at the atomic level, i.e. only constructs without logical connectives have to be negated. If the respective atom is an attribute comparison, a component or a connection atom then we can directly use classical negation in ASP. In case the atom represents an arithmetic expression we eliminate negation by transforming the expression, i.e. by changing the arithmetic operator, for example:

$$\neg(3a + 5b \leq c) \quad \Longrightarrow \quad 3a + 5b > c$$

Alternatively, we support a second approach by simply interpreting negation in the axiomatization as default negation. This basically means that the semantics of the axiomatization in terms of negation is defined by the chosen target language and not by the axiomatization itself. For an overview on the topic of default vs. classical negation, see the fundamental paper by Gelfond and Liftschitz [Gelfond and Lifschitz, 1991].

Disjunction semantics

Note also that our translation does not rely upon disjunctive ASP dialects, i.e. ASP extensions for modelling inclusive/exclusive disjunctions. A well-known problem with the ASP stable model semantics is that inclusive disjunction could be falsely interpreted as exclusive disjunction because of the minimal model interpretation. Disjunction in ASP is neither strictly inclusive nor exclusive but subject to minimization. By using cardinality rules with appropriate bounds instead we are able to enforce the intended disjunction

semantics in (exclusive-or) one-to-many connections. An inclusive disjunction can be modelled with a cardinality rule by setting lower bound $l = 1$ and upper bound $u = n$, with n being the number of disjunctive components; similarly, an exclusive disjunction can be obtained by setting $l = 1$ and $u = 1$. Disjunctions in the ϕ constraint subformulas are generally inclusive.

Complexity of reasoning

Let us briefly comment on the complexity of the relevant reasoning tasks in ASP: Deciding satisfiability of answer set programs is NExpTime-complete [Dantsin et al., 2001] if programs contain logical variables. Hence this task is probably slightly harder than deciding LoCo satisfiability. For ground answer set programs (i.e. programs where logical variables have been substituted by ground terms in an equivalence preserving manner) deciding satisfiability is NP-complete [Simons et al., 2002] — the grounding algorithms incur an exponential blowup.

5.3 Transformation to MiniZinc

This section deals with a presentation on how to transform the main LoCo language elements to MiniZinc on an abstract level. MiniZinc is a medium-level constraint modelling language gaining a lot of interest recently for its aim of becoming a standard modelling language for the constraint programming community [Nethercote et al., 2007]. It supports solver-independent modelling by compiling high-level user-defined methods into low-level solver methods, so in a way it has a similar workflow than our LoCo approach. Currently more and more state-of-the-art solvers from the integer programming, Sat or constraint programming community provide interfaces to MiniZinc s.t. a transformation from LoCo to MiniZinc is very promising since it would instantly provide us with a whole portfolio of industry-level solvers as potential target languages.

5.3.1 Representing Components

Component Usage

We represent whether a component is used or not via a Boolean variable:

```
array[1..n] of var bool : c-id-used;
```

We write `c-id-used[i]=true` to express the fact that component instance i of component c is used in the model. For the input components we include for all n:

```
constraint c-id-used[n] = true;
```

Component Catalogue

Next we show how to include an extensional component catalogue. Of course, the catalogue can just as well be described by any combination of constraints available/definable in MiniZinc. For each component type we include a 2D array of the possible component instantiations. As above let n denote the maximum available number of components of type C.

```
% Components attributes
set of int : ATTRIBUTES = 1..noofattributes;
% Naming the attributes
% Can later write c-components[i,attr2name] to
% access the attr2name attribute (2nd attribute) of component instance i
int : attr1name = 1;
int : attr2name = 2;
...
int : attrnoofattributesname = noofattributes;

% No of catalogue entries
set of int : ENTRIES = 1..cataloguesize;

% The catalogue in extension
array[ENTRIES,ATTRIBUTES] of int : catalogue =
```

```
    [ firsttuple |
      nexttuple |
      ...
      lasttuple ];

% A null-tuple
array[ATTRIBUTES] of 0 : zeroentry;

% An array of the available components
% e.g.: access attribute name of instance i of component c
% by writing c-components[i,name]
array[1..n,ATTRIBUTES] of var int : c-components;

% Each component conforms to some catalogue entry
% (or is unused)
constraint
   forall(i in 1..n)
      ( ( c-id-used[i] = true ->
            table(c-components[i,_],catalogue) )
        /\
        ( c-id-used[i] = false ->
            table(c-components[i,_],zeroentry) ) );
```

Note that the above model allows integer-valued component attributes only. In order to represent enumerations we would for example need to introduce an intermediate mapping structure from integers to strings. Since the main purpose of this section is not completeness but to show the basic ideas on how to transform the main LoCo language elements we will leave it with attributes being of type integer.

5.3.2 Representing Connections

We now turn to connections between components. For the moment we ignore constraints on the connections — all we are interested in is how to represent the cardinality constraints arising from LoCo's various connection axioms. We represent connections between components via a matrix-model. Assume component type C_1 can be connected to

component type C_2 and $|C_1| = n$ as well as $|C_2| = m$. We use 0-1-integer variables for $1 \leq i \leq n$ and $1 \leq j \leq m$.[1]

Binary Connections

A binary connection between C_1 and C_2 is then modelled via a matrix:

```
array[1..n,1..m] of var int 0..1 : c1-to-c2;
```

For the lower and upper bounds LB, UB on the number of connections between any one given C_1 and all of the C_2 for each $1 \leq i \leq n$ we introduce the respective count variables and constraints:

```
array[1..n] of var LB..UB : c1-to-c2-count;
constraint
   forall(i in 1..n)(c1-id-used[i] = true ->
      count_eq(c1-to-c2[i,_], 1, c1-to-c2-count[i]) );
```

If C_1 is not an input component we also add the constraint:

```
constraint
   forall(i in 1..n) (member(c1-to-c2[i,_], 1) ->
      c1-id-used[i] = true);
```

Now repeat all of the above for the lower and upper bounds in the direction from C_2 to C_1 (swapping rows and columns in the matrix).

Self Connections Self connections are special in that the matrix is now symmetric:

```
constraint
   forall(i, j in 1..n where i < j) (c-to-c[i,j] = c-to-c[j,i]);
```

[1] If we use Booleans instead we lose a lot of global constraints.

Unfolded Binary Connections For unfolded binary connections with multiple cases (say m) we modify the model as follows:

```
array[1..n,1..m] of var int : c1-to-c2-count;
array[1..m] of int : m-lbs = [lb1,...,lbm];
array[1..m] of int : m-ubs = [ub1,...,ubm];
constraint
   forall(i in 1..m)
      ( forall(j in 1..n)
         ( c1-to-c2-count[i,j] >= m-lbs[i] /\
           c1-to-c2-count[i,j] =< m-ubs[i] ) );
constraint
   forall(i in 1..n)( exists (j in 1..m)
      ( c1-id-used[i] = true ->
         (count_eq(c1-to-c2[i,_], 1, c1-to-c2-count[i,j]) ) ) );
```

We may assume at this point that the m cases are mutually exclusive so that the "exists" (which becomes an ordinary disjunction) is ok.

One-to-many Connections

For one-to-many connections we reuse the 0-1-integer variables we introduced for the binary connections.

Standard One-to-many Connections Assume there are up to n instances of component type C. Let m be the sum of the upper bounds on the number of component instances appearing on the right hand side of the axiom. We use a $n \times m$ matrix. Assume that `c-to-ci` is the name of the binary connection matrix between C and C_i. By r we denote the number of different component types on the right hand side.

```
array[1..n,1..m] of var int 0..1 : c-to-cs =
   [ c-to-c1[1,1], c-to-c2[1,2], ..., c-to-ci[1,r-max]|
     ...,
     c-to-c1[n,1], c-to-c2[n,2], ..., c-to-ci[n,r-max]];
```

The counting constraints are similar to the binary connections:

```
array[1..n] of var LB..UB : c-to-cs-count;
constraint
   forall(i in 1..n)(c-id-used[i] = true ->
      count_eq(c-to-cs[i,_], 1, c-to-cs-count[i]) );
```

There is no need to post constraints that activate `c-id-used`-variables; this is done in the binary connection axioms.

Exclusive-OR One-to-many Connections Next there are one-to-many connections of the "exclusive or" variant:

```
% As before
array[1..n] of var LB..UB : c-to-cs-counts;
% The indices of the different component types in the
% matrix, one per type, 1 <= j <= r
set of int : c-j-range = l..u;
...
% The constraint
constraint
   forall(i in 1..n)(c-id-used[i] = true ->
      ( count_eq(c-to-cs[i,_], 1, c-to-cs-count[i]) /\
% Include the below for all c-j-range
% (We use at least one c-j, hence...)
        ( member(c-to-cs[i,c-j-range],1) ->
% Use conjunction over all c-notj-range
% (... we cannot use any c-notj)
            ( count_eq(c-to-cs[i,c-notj-range],1,0) ) ) ) );
```

Generalized One-to-many Connections Assume that there are r different components mentioned on the right hand side of the axiom, and that the axiom has c cases. Assume there are at most n components of type C. Let's introduce a 2D matrix of the cardinalities mentioned in the axiom. We also introduce an array of r count variables for each component of type C; these are tabled over the 2D matrix.

```
% The cardinalities in the axiom
array[1..c,1..r] of int : c-to-cs-cards =
  [ firsttuple |
    nexttuple |
    ...
    lasttuple ];
% A tuple of r zeros
array[1..r] of 0 : zerotuple;
% For each component a tuple of r cardinality variables
array[1..n,1..r] of var int : c-to-cs-counts;
% The array of 0-1 connection variables
array[1..n,1..m] of var int 0..1 : c-to-cs =
   [ c-to-c1[1,1], c-to-c1[1,2], ..., c-to-ci[1,r-max]|
     ...,
     c-to-c1[n,1], c-to-c1[n,2], ..., c-to-ci[n,r-max]];
% Count the respective 1-entries in the above matrix
% The indices of the different component types in the
% matrix, one per type, 1 <= j <= r
set of int : c-j-range = l..u;
...
% The counting constraints
constraint
   forall(i in 1..n)
% Include the below for all c-j-range
      ( count_eq(c-to-cs[i,c-j-range], 1, c-to-cs-count[i,j]) /\
        ... /\
        count_eq(c-to-cs[i,c-j-range], 1, c-to-cs-count[i,j]) );
% Ensure the 1-entries adhere to one of the axiom's cases
constraint
   forall(i in 1..n)
      ( ( c-id-used[i] = true ->
            table(c-to-cs-counts[i,_], c-to-cs-cards) )
        /\
        ( c-id-used[i] = false ->
            table(c-to-cs-counts[i,_], zerotuple) ) );
```

5.3.3 Adding the Constraints on Connections

Recall that for each component type we have a matrix `c-components` with one entry per component instance (not all of which have to be used). For each instance there is a row in the matrix, representing its attributes. We can access an attribute `name` of component i by writing `c-components[i,name]`. For each instance there is a Boolean variable `c-id-used[i]` indicating whether that instance is used in the configuration.

MiniZinc supports logical combinations of constraints, in particular using the connectives `/\, \/, ->, <-, <->` and `not`.
For the constraints $\phi(\vec{x})$ in LoCo's axioms we have to consider two cases:

(1) The $\vec{x}$ occur in some aggregate like `sum, count`.

(2) The $\vec{x}$ appear in some abbreviated conjunction of atomic constraints like $\vec{x} < 5$ (i.e. $\bigwedge_i x_i < 5$).

Binary Connections

Assume we have a binary connection axiom with MiniZinc matrix `c1-to-c2`. Let the constraint be $\phi(\vec{x}, \vec{y})$ with $\vec{x}$ the attributes of C_1 and $\vec{y}$ the attributes of C_2. For the Boolean structure of ϕ we use MiniZinc's Boolean connectives. On the atomic level we distinguish two cases: For case (1) we use a global constraint such as

```
( c-id-used[i] ->
   count_eq([c1-components[i,name],
             c2-components[1,name]*c1-to-c2[i,1],
             ...,
             c2-components[m,name]*c1-to-c2[i,m]],
            5) );
```

By multiplying C_2's attributes with the variable representing the connection between C_1 and C_2 we filter out those component instances that are not connected to this C_1.
For case (2) we use a conjunction of primitive constraints such as

```
( c-id-used[i] ->
   (c1-components[i,name] < 5 /\
    c2-components[1,name]*c1-to-c2[i,1] < 5 /\
    ...,
    c2-components[m,name]*c1-to-c2[i,m] < 5) );
```

The resulting translation of ϕ to MiniZinc then has to be added to the counting constraint on the matrix representing the binary connection (this also is where the i comes from). Note that this renders the use of the conditional `c-id-used[i]` superfluous.

One-to-many Connections

For one-to-many connections the story is very similar. The major difference is that we need to address different component types in the translation on the atomic level. Again the formulas have to be added to the counting constraints on the connection matrices.

5.3.4 Adding Partial Configurations

Partial configurations to be used/avoided in the final configuration can be enforced by setting the respective connection or attribute variables (positive information) or posting disequality constraints (negative information).

Chapter 6

Evaluation

6.1 Industrial benchmark problems

The evaluation focuses on two problems we got from our industrial partner Siemens of which both have highly practical relevance. The House Problem has been introduced in section 1.3 and has been our running example throughout this thesis. Since the problem has now been discussed in large parts even down to the representation of its main constraints there is no need for another problem description at this point anymore. The interested reader can find a detailed problem analysis in [Bettex, 2009].

The second practical problem we received from Siemens is the Partner Units Problem (PUP) presented in [Falkner et al., 2011]. Next to modelling the problem in LoCo for benchmark purposes we have spent substantial time and effort in a deeper analysis of this problem with theoretical and practical results published in [Aschinger et al., 2011a, Aschinger et al., 2011c, Aschinger et al., 2011d]. For this reason we present the problem in more detail and give a short overview of the main research results.

The PUP problem originated from configuring railway interlocking systems but is not limited to this application domain and has also widespread practical relevance in other domains such as security and surveillance systems. It captures the essence of a specific type of configuration problem that frequently occurs in industry and occurs whenever sensors that are grouped into zones have to be attached to control units and the communication between units should be kept as simple and effective as possible.

Typical applications include intelligent traffic management or surveillance and security applications. It has recently also been introduced as a benchmark problem in the Third Answer Set Programming Competition [ASP Competition, 2011] where it turned out to be one of the most difficult problems to solve efficiently.

Informally the PUP can be described as follows: Consider a set of sensors that are grouped into zones. A zone may contain many sensors and a sensor may be attached to more than one zone. The task of the PUP then is to connect the sensors and zones to control units, where each control unit can be connected to the same fixed maximum number *UnitCap* of zones and sensors. Moreover, if a sensor is attached to a zone, but the sensor and the zone are assigned to different control units, then the two control units in question have to be directly connected. However, a control unit cannot be connected to more than *InterUnitCap* other control units (the partner units).

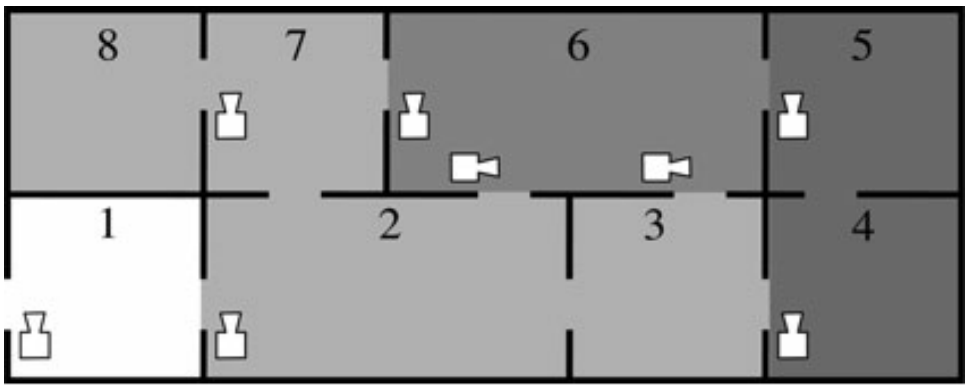


Figure 6.1: Room layout for a PUP scenario

The PUP occurs e.g. in the following application domain: Consider a museum where we want to keep track of the number of visitors that populate certain parts (zones) of the building. The doors leading from one zone to another are equipped with sensors. To keep track of the visitors the zones and sensors are attached to control units; the adjacency constraints on the control units ensure that communication between units can be kept simple. Figure 6.1 shows a potential room layout with 8 rooms. Most rooms are equipped with a sensor. For example, there is a sensor between rooms 1 and 2 but not between 2 and 3. Rooms are grouped into zones, e.g. zones Z1 (white), Z2378 (light

gray) and Z45 (dark gray). The number sequence represents the rooms involved in a zone. Zones can also overlap, i.e. a room potentially belongs to more than one zone. The relationship between zones and sensors of this example can be depicted in the form of a bipartite graph (6.2).

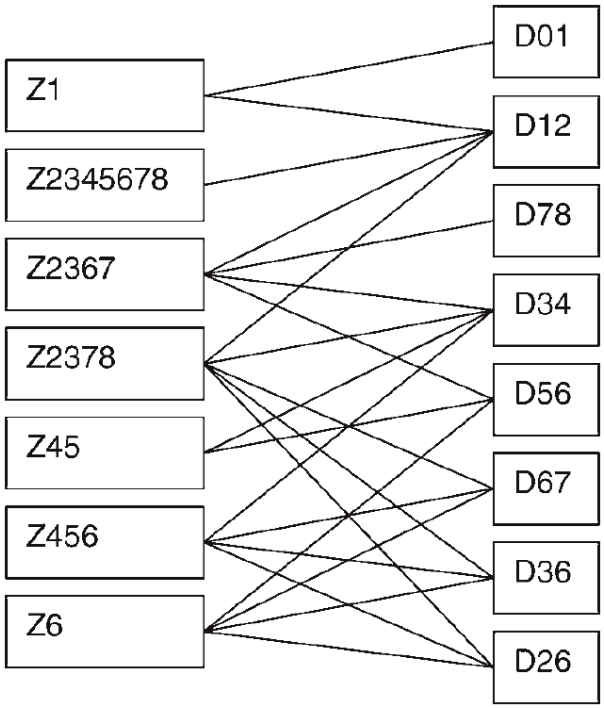


Figure 6.2: Bipartite graph representation of the room layout from figure 6.1

This representation is the cornerstone of the formal definition: The PUP consists of partitioning the vertices of a bipartite graph $G = (V_1, V_2, E)$ into a set U of bags such that each bag

- contains at most *UnitCap* vertices from V_1 and at most *UnitCap* vertices from V_2; and
- has at most *InterUnitCap* adjacent bags where the bags U_1 and U_2 are adjacent whenever $v_i \in U_1$ and $v_j \in U_2$ and $(v_i, v_j) \in E$.

To every solution of the PUP we can associate a solution graph. For this we associate to

every bag $u \in U$ a vertex $u' \in U'$. Then the solution graph G^* has the vertex set $V_1 \cup V_2 \cup U'$ and the set of edges $\{(v, u') \mid v \in u \wedge u \in U\} \cup \{(u'_i, u'_j) \mid u_i \text{ and } u_j \text{ are adjacent.}\}$.

Figure 6.3 shows a PUP instance and a solution for the case *UnitCap=InterUnitCap=2*: six sensors (left) and six zones (right) which are completely inter-connected are partitioned into units - shown as squares - respecting the adjacency constraints. Note that for the given parameters this is a maximal solvable instance; it is not possible to connect a new zone or sensor to any of the existing ones.

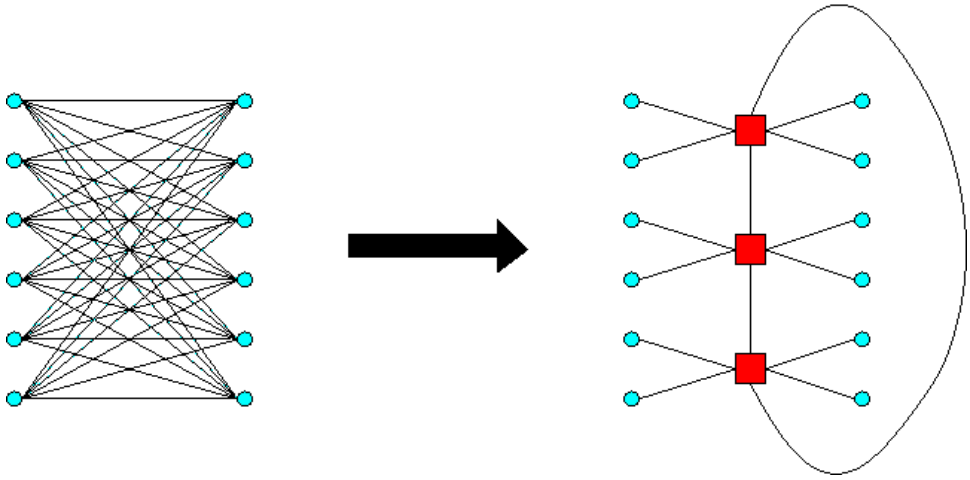


Figure 6.3: Partitioning of a $K_{6,6}$ Partner Units Instance

In [Aschinger et al., 2011d] we have shown that the case where $\mathit{InterUnitCap} = 2$ and $\mathit{UnitCap} = k$ for some fixed k is tractable by giving a specialized NLOGSPACE algorithm that is based on the notion of path decomposition. This special case is of great interest to our industrial partner. The paper includes a detailed description of the mentioned algorithm and shows that it can find optimal solutions much faster than a standard CSP encoding of the PUP. We have also been working on encodings of the general version of the PUP - that is where both *UnitCap* and *InterUnitCap* are arbitrary fixed constants - in the frameworks of answer set, integer and constraint programming as well as SAT solving [Aschinger et al., 2011a]. Both of these papers also include a number of complexity results for other special cases of the problem.

6.2 Benchmarks

We have evaluated our LoCo encodings of both problems on a set of benchmark instances that we received from our industrial partners.[1] We compare the runtimes of our LoCo implementation against previously developed hand-crafted ASP encodings. The experimental results that we obtain are very encouraging: For the House Problem we can compete with the hand-written problem encoding in answer set programming presented in [Friedrich et al., 2011]; it turns out that our translation of the declarative LoCo specification yields a very similar program.

Problem instance	Search		Optimization	
	LoCo enc.	Manual enc.	LoCo enc.	Manual enc.
p02t06	0.03	0.03	0.1	0.03
p02t10	0.05	0.03	0.12	0.05
p03t15	0.07	0.04	0.22	0.35
p04t20	0.13	0.05	0.65	0.5
p05t25	0.18	0.06	1.33	0.75
p10t50	0.92	0.21	X	X
p15t75	2.95	0.51	X	X
p20t100	7.14	1.19	X	X
p30t150	24.72	4.35	X	X
p40t200	65.15	25.92	X	X

Table 6.1: Benchmarks for the House Problem

Table 6.1 shows the benchmark results for the House Problem. All experiments were conducted on a 2.5GHz Intel Core2 Quad CPU with 4 GB RAM running Windows7 64-bit. In general we have imposed a ten minute time limit for finding solutions in our experiments. In this evaluated variant of the House Problem persons and things are set to be input components and the task is to find the minimal number of needed cabinets and rooms. The problem instances differ only with regard to the number of given persons and things involved and hence to the size of the search space.

We have evaluated the instances in two different settings: (1) the original setting (Optimization) and (2) a simplified version (Search) where the effectively needed number

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

of all components is already part of the input. In the latter case the search for the minimal number of needed components is eliminated and the configuration problem reduces to connecting components correctly with regard to the side constraints. In general the outcome of our experiments can be summarized as follows: The LoCo encoding performs slightly worse than the manual encoding in both settings. This overhead probably has to be attributed to the additional helper predicates created when transforming the constraint parts of the rules. In the easier "Search" setting all instances are solvable whereas in the "Optimization" setting both the LoCo and the manual encoding run into timeouts for the same instances.

Problem instance	Search		Optimization	
	LoCo enc.	Manual enc.	LoCo enc.	Manual enc.
small-7	0.1	0.1	0.13	0.11
small-8	0.1	0.1	0.21	0.15
single-11	0.15	0.1	0.19	0.4
small-no	1.18	0.95	33.05	70.16
double-10	0.15	0.08	0.35	0.19
double-14	0.77	0.19	10.63	1.6
double-16	1.32	0.53	X	379.64
double-20	44.15	1.52	X	X

Table 6.2: Benchmarks for the Partner Units Problem

We have evaluated the LoCo version of the PUP and were able to reach about the same performance as the manually written answer set program presented in [Aschinger et al., 2011a] if for the latter the problem-specific search strategy is turned off (see table 6.2 for results). In particular, the gap between the automatic translation and the hand-written problem encoding is similar to what we have experienced for the House Problem which further justifies our trust in the quality of the automated LoCo transformations. [Aschinger et al., 2011a] further compares the ASP version of the PUP with SAT, CSP and integer programming encodings as well as with a Java-based implementation optimized for a special case. Since all these encodings are using problem-specific solving heuristics we are not taking them into account for the benchmarks presented here.

Chapter 7

Conclusion

7.1 Summary and main results

The main scientific contribution of LoCo as a whole to the area of configuration research can be summarized as follows: Like conditional or generative Csp LoCo supports the conditional inclusion of components into configurations as a knowledge representation idiom. In contrast to competing approaches, in LoCo the number of available components does not have to be specified manually for all component types involved; yet LoCo does not face termination issues. Apart from this, LoCo features ports, a means to describe arbitrary component connection layouts as well as a rich language for describing constraints on the admissible combinations. Moreover, the user can specify partial configurations (not) to be used for building the configuration. Our prototypical implementation in answer set programming has proven that our approach is applicable in practice. Benchmark results have shown that the automatically generated output code for practically relevant problem scenarios is competitive. Note that the main intention here is not to compete with highly specialized problem-specific encodings but rather on the knowledge representation level in order to provide a way to formalise the configuration knowledge on a high level without the need of any solver-specific details. The final results of this project have been published in the ACM Transactions on Computational Logic [Aschinger et al., 2014].

7.2 Future Work

Let us conclude by pointing out promising directions for future research. On the theoretical side it would be nice to determine the complexity of deciding LoCo satisfiability in the case of extensional component catalogues. Likewise it would be interesting to determine fragments of LoCo where this question is tractable. We observe that a starting point for this could be the tractable fragments of the configuration logic introduced in [Gottlob et al., 2007] — a LoCo fragment restricted to input components only.

In order to increase the practical usability of LoCo it would be helpful to either develop a graphical user interface or to single out a fragment/extension of UML and OCL corresponding to LoCo. On the reasoning side there also remain a number of challenges: On the one hand side, the implementation via answer set programming is now mature, and a detailed description thereof together with a thorough experimental evaluation is currently underway. On the other hand, we would like to have complementary reasoning methods at our disposal. To this end we have theoretically developed a translation into the MiniZinc language [Nethercote et al., 2007], providing us with access to a whole portfolio of integer programming, Sat and constraint solvers; however, an implementation is still missing. It would also be promising to develop an implementation that follows the idea of incrementally adding components instead of pre-generating all potentially useful ones; here, the absence of freely available solvers for generative or conditional Csp constitutes an additional obstacle.

Concerning possible extensions of LoCo, first on the list are adding support for explanation as well as for computing optimal configurations. Support for reconfiguration just as well as the integration with a classification-based configuration formalism appear to be considerably more challenging: The former because of the need for dealing with inconsistencies in a logical framework, the latter because of the apparent need for some kind of inheritance relation between the constraints attached to connections.

Bibliography

[Amilhastre et al., 2002] Amilhastre, J., Fargier, H., and Marquis, P. (2002). Consistency Restoration and Explanations in Dynamic CSPs—Application to Configuration. *Artificial Intelligence*, 135(1–2):199–234.

[Andersen et al., 2010] Andersen, H. R., Hadzic, T., and Pisinger, D. (2010). Interactive cost configuration over decision diagrams. *Journal of Artificial Intelligence Research (JAIR)*, 37:99–139.

[Aschinger et al., 2011a] Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., and Thorstensen, E. (2011a). Optimization methods for the partner units problem. In *Proceedings of the 8th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011)*, Lecture Notes in Computer Science, Berlin, Germany.

[Aschinger et al., 2011b] Aschinger, M., Drescher, C., and Gottlob, G. (2011b). Introducing LoCo, a Logic for Configuration Problems. In *Proceedings of the 2nd Workshop on Logics for Component Configuration, LoCoCo 2011*, Perugia, Italy.

[Aschinger et al., 2011c] Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., and Thorstensen, E. (2011c). Structural decomposition methods, and what they are good for. In *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, Dortmund, Germany. Invited Paper.

[Aschinger et al., 2011d] Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., and Thorstensen, E. (2011d). Tackling the partner units configuration problem. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, Barcelona, Spain.

[Aschinger et al., 2014] Aschinger, M., Drescher, C., Gottlob, G., and Vollmer, H. (2014). LoCo — A Logic for Configuration Problems. *ACM Transactions on Computational Logic*. Accepted for publication.

[Aschinger et al., 2012] Aschinger, M., Drescher, C., and Vollmer, H. (2012). LoCo — A Logic for Configuration Problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 73–78. IOS Press.

[Aschinger et al., 2010] Aschinger, M., Jessenitschnig, M., and Zanker, M. (2010). Constraint-based personalized configuring of product and service bundles. *International Journal on Mass Customization*, 3(4):407–425.

[ASP Competition, 2011] ASP Competition (2011). Third International Answer Set Programming Competition.
https://www.mat.unical.it/aspcomp2011/.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F. (2003). *The Description Logic Handbook: Theory, Implementation, Applications*. Cambridge University Press, Cambridge, UK.

[Bettex, 2009] Bettex, M. (2009). House configuration problem using constraint optimization. Master's thesis, School of Computer and Information Science, University of South Australia, Adelaide, Australia.

[Bettex et al., 2009] Bettex, M., Falkner, A., Mayer, W., and Stumptner, M. (2009). On Solving Complex Rack Configuration Problems using CSP Methods. In *Configuration Workshop at the 21st International Conference on Artificial Intelligence (IJCAI)*, pages 53–60, Pasadena, California. AAAI Press.

[Borgida et al., 1989] Borgida, A., Brachman, R., McGuinness, D., and Resnick, L. (1989). Classic: A structural data model for objects. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 59 – 67, Portland, Oregon.

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

[Brown and Chandrasekaran, 1989] Brown, D. and Chandrasekaran, B. (1989). *Design Problem Solving: Knowledge Structures and Control Strategies*. Pitman Publishing Ltd., London, UK.

[Brown, 1998] Brown, D. C. (1998). Defining configuring. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (AI EDAM), Special Issue: Configuration Design*, 12(4):301 – 305.

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

[Chandra et al., 1981] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. (1981). Alternation. *Journal of the ACM*, 28(1):114–133.

[Chandrasekaran, 1990] Chandrasekaran, B. (1990). Design problem solving. *AI Magazine*, 11(4):59 – 71.

[COIN-OR, 2014] COIN-OR (2014). COIN-OR - COmputational INfrastructure for Operations Research. http://www.coin-or.org/.

[Cunis et al., 1989] Cunis, R., Günter, A., Syska, I., Peters, H., and Bode, H. (1989). PLAKON - an approach to domain-independent construction. In *Proceedings of the Second International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IAE/AIE)*, pages 866 – 874.

[Dantsin et al., 2001] Dantsin, E., Eiter, T., Gottlob, G., and Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425.

[Dowling and Gallier, 1984] Dowling, W. F. and Gallier, J. H. (1984). Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 1(3):267–284.

[Enderton, 1972] Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press.

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

[Falkner et al., 2011] Falkner, A., Haselböck, A., Schenner, G., and Schreiner, H. (2011). Modeling and solving technical product configuration problems. *AI EDAM*, 25(2):115–129.

[Falkner and Schreiner, 2014] Falkner, A. and Schreiner, H. (2014). Siemens: configuration and reconfiguration in industry. In Felfernig, A., Hotz, L., Bagley, C., and Tiihonen, J., editors, *Knowledge-based Configuration - From Research to Business Cases*, pages 199 – 210. Morgan Kaufmann, Waltham, MA.

[Feinerer, 2013] Feinerer, I. (2013). Efficient large-scale configuration via integer linear programming. *AI EDAM*, 27(1):37–49.

[Fleischanderl et al., 1998] Fleischanderl, G., Friedrich, G., Haselböck, A., Schreiner, H., and Stumptner, M. (1998). Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68.

[Friedrich et al., 2011] Friedrich, G., Ryabokon, A., Falkner, A. A., Haselböck, A., Schenner, G., and Schreiner, H. (2011). (re)configuration based on model generation. In Drescher, C., Lynce, I., and Treinen, R., editors, *LoCoCo*, volume 65 of *EPTCS*, pages 26–35.

[Friedrich and Stumptner, 1999] Friedrich, G. and Stumptner, M. (1999). Consistency-Based Configuration. In *Configuration Workshop at the 16th National Conference on Artificial Intelligence (AAAI)*, pages 35–40, Orlando, Florida. AAAI Press.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability*. W.H. Freeman and Co., New York.

[Gebser et al., 2011] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Schneider, M. (2011). Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, 24(2):105–124.

[Gebser et al., 2008] Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. (2008). Engineering an incremental ASP solver. In de la Banda, M. G. and Pontelli, E., editors, *Logic Programming, 24th International Conference, ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205, Udine, Italy. Springer.

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

[Gelfond, 2008] Gelfond, M. (2008). Answer Sets. In van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*, pages 285 – 316. Elsevier.

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

[Gelle and Weigel, 1996] Gelle, E. and Weigel, R. (1996). Interactive configuration using constraint satisfaction techniques. In *In Second International Conference on Practical Application of Constraint Technology, PACT-96*, pages 37–44. Menlo Park, AAAI Press.

[Gottlob et al., 2007] Gottlob, G., Greco, G., and Mancini, T. (2007). Conditional constraint satisfaction: Logical foundations and complexity. In *IJCAI'07*.

[Hentenryck, 1999] Hentenryck, P. V. (1999). *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA.

[Hermann and Sertkaya, 2008] Hermann, M. and Sertkaya, B. (2008). On the Complexity of Computing Generators of Closed Sets. In *Proceedings of the 6th International Conference on Formal Concept Analysis ICFCA'08*, pages 158–168, Montreal, Canada. Springer.

[Hubaux et al., 2012] Hubaux, A., Jannach, D., Drescher, C., Murta, L., Mannisto, T., Czarnecki, K., Heymans, P., Nguyen, T., and Zanker, M. (2012). Unifying software and product configuration: A research roadmap. In *Proceedings of the ECAI 2012 Workshop on Configuration*, pages 31–35, Montpellier, France. CEUR-WS.

[JavaCC, 2014] JavaCC (2014). JavaCC - Java Compiler Compiler. https://javacc.java.net/.

[Johnson et al., 1988] Johnson, D. S., Yannakakis, M., and Papadimitriou, C. H. (1988). On Generating All Maximal Independent Sets. *Information Processing Letters*, 27:119–123.

[Junker, 2004] Junker, U. (2004). QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, AAAI'04*, pages 167–172, San Jose, California, USA. AAAI Press / The MIT Press.

[Junker, 2006] Junker, U. (2006). Configuration. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, pages 837 – 874. Elsevier.

[Karatas et al., 2010] Karatas, A. S., Oguztüzün, H., and Dogru, A. H. (2010). Global constraints on feature models. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming, CP 2010*, pages 537–551, St. Andrews, Scotland, UK. Springer.

[Kiziltan and Hnich, 2001] Kiziltan, Z. and Hnich, B. (2001). Symmetry breaking in a rack configuration problem. In *Proceedings of the IJCAI-2001 Workshop on Modelling and Solving Problems with Constraints*, Seattle, Washington.

[Kolaitis and Vardi, 1998] Kolaitis, P. G. and Vardi, M. Y. (1998). Conjunctive-query containment and constraint satisfaction. In *PODS'98*.

[Lenzerini and Nobili, 1990] Lenzerini, M. and Nobili, P. (1990). On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems*, 15(4):453 – 461.

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

[Lucchesi and Osborn, 1978] Lucchesi, C. L. and Osborn, S. L. (1978). Candidate Keys for Relations. *Journal of Computer and System Sciences*, 17(2):270–279.

[Mailharro, 1998] Mailharro, D. (1998). A classification and constraint-based framework for configuration. *AI EDAM*, 12(4):383–397.

[McDermott, 1982] McDermott, J. (1982). R1: A Rule-based Configurer of Computer Systems. *Artificial Intelligence*, 19:39–88.

[McGuinness, 2003] McGuinness, D. L. (2003). Configuration. In Baader, F., Calvanese, D., McGuinness, D., and Patel-Schneider, D. N. P., editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 397 – 414. Cambridge University Press.

[McGuinness and Wright, 1998] McGuinness, D. L. and Wright, J. R. (1998). Conceptual modelling for configuration: A description logic-based approach. *AI EDAM*, 12(4):333–344.

[Mittal and Falkenhainer, 1990] Mittal, S. and Falkenhainer, B. (1990). Dynamic Constraint Satisfaction Problems. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI)*, pages 25–32, Boston, Massachusetts. AAAI Press / The MIT Press.

[Mittal and Frayman, 1989] Mittal, S. and Frayman, F. (1989). Towards a generic model of configuration tasks. In *Proceedings of the 11th International Conference on Artificial Intelligence (IJCAI)*, pages 1395 – 1401, Detroit, Michigan. Morgan Kaufmann.

[Mouhoub and Sukpan, 2007] Mouhoub, M. and Sukpan, A. (2007). Solving conditional and composite constraint satisfaction problems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*, pages 336–337, New York, NY, USA. ACM.

[Nardi and Brachman, 2003] Nardi, D. and Brachman, R. (2003). An introduction to description logics. In Baader, F., Calvanese, D., McGuinness, D., and Patel-Schneider, D. N. P., editors, *The Description Logic Handbook: Theory, Implementation and Applications*, pages 5 – 44. Cambridge University Press.

[Nethercote et al., 2007] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. (2007). MiniZinc: Towards a Standard CP Modelling Language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming CP, Providence, RI.*

[Niemelä et al., 1999] Niemelä, I., Simons, P., and Soininen, T. (1999). Stable model semantics of weight constraint rules. In Gelfond, M., Leone, N., and Pfeifer, G., editors, *LPNMR*, volume 1730 of *Lecture Notes in Computer Science*, pages 317–331. Springer.

[Sabin and Freuder, 1996] Sabin, D. and Freuder, E. C. (1996). Configuration as Composite Constraint Satisfaction. In *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop (AIMRP)*, pages 153 – 161, Albuquerque, New Mexico. AAAI Press.

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

[Schneeweiss and Hofstedt, 2011] Schneeweiss, D. and Hofstedt, P. (2011). Fdconfig: A constraint-based interactive product configurator. In *19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011)*, pages 239–255. Springer.

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

[Sinz et al., 2003] Sinz, C., Kaiser, A., and Küchlin, W. (2003). Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97.

[Soininen and Gelle, 1999] Soininen, T. and Gelle, E. (1999). Dynamic constraint satisfaction in configuration. In *Papers from the AAAI Workshop on Configuration*, pages 95 – 100. AAAI Press.

[Soininen and Niemelä, 1998] Soininen, T. and Niemelä, I. (1998). Developing a declarative rule language for applications in product configuration. In Gupta, G., editor, *Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer Berlin / Heidelberg.

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

[Stumptner and Haselböck, 1993] Stumptner, M. and Haselböck, A. (1993). A generative constraint formalism for configuration problems. In *Proceedings of the Third Congress of the Italian Association for Artificial Intelligence on Advances in Artificial Intelligence*, AI*IA '93, pages 302–313, London, UK. Springer-Verlag.

[Stumptner et al., 1994] Stumptner, M., Haselböck, A., and Friedrich, G. (1994). COCOS - a tool for constraint-based, dynamic configuration. In *Proceedings of the 10th Conference on Artificial Intelligence for Applications*, pages 373–380, San Antonio, TX, USA.

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

[ECL^{i}PS^e, 2014] ECL^{i}PS^e (2014). ECL^{i}PS^e-Prolog. http://eclipseclp.org/.

[Thorstensen, 2010] Thorstensen, E. (2010). Capturing configuration. In *Doctoral Program at CP'10*.

[Tsang, 1993] Tsang, E. P. (1993). *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego.

[Wielinga and Schreiber, 1997] Wielinga, B. J. and Schreiber, G. (1997). Configuration-design problem solving. *IEEE Intelligent Systems*, 12(2):49 – 56.

[Wright et al., 1995] Wright, J., McGuinness, D., Foster, C., and Vesonder, G. (1995). Conceptual modeling using knowledge representation: Configurator applications. In *Proceedings of the Artificial Intelligence in Distributed Information Networks Workshop, IJCAI-95*, Montreal, Canada.

[Wright et al., 1993] Wright, J. R., Weixelbaum, E., Vesonder, G. T., Brown, K. E., Palmer, S. R., Berman, J. I., and Moore, H. H. (1993). A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&t network systems. *AI Magazine*, 14(3):69 – 80.

Appendix A

Publications

- Aschinger, M., Drescher, C., Gottlob, G., and Vollmer, H. (2014). LoCo – A Logic for Configuration Problems. In *ACM Transactions on Computational Logic*. Accepted for publication.

- Aschinger, M., Drescher, C., and Vollmer, H. (2012). LoCo – A Logic for Configuration Problems. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 7378. IOS Press.

- Aschinger, M., Drescher, C., and Gottlob, G. (2011). Introducing LoCo, a Logic for Configuration Problems. In *Proceedings of the 2nd Workshop on Logics for Component Configuration, LoCoCo 2011*, Perugia, Italy.

- Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., and Thorstensen, E. (2011). Structural decomposition methods, and what they are good for. In *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, Dortmund, Germany. Invited Paper.

- Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., and Thorstensen, E. (2011). Optimization methods for the partner units problem. In *Proceedings of the 8th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming for*

Combinatorial Optimization Problems (CPAIOR 2011), Lecture Notes in Computer Science, Berlin, Germany.

- Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., and Thorstensen, E. (2011). Tackling the partner units configuration problem. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, Barcelona, Spain.