(Re)configuration using Answer Set Programming*

Gerhard Friedrich and
Anna Ryabokon
Universitaet Klagenfurt, Austria
firstname.lastname@aau.at

Andreas A. Falkner, Alois Haselböck,
Gottfried Schenner and **Herwig Schreiner**
Siemens AG Österreich, Vienna, Austria
firstname.{middleinitial.}lastname@siemens.com

Abstract

Reconfiguration is an important activity for companies selling configurable products or services which have a long life time. However, identification of a set of required changes in a legacy configuration is a hard problem, since even small changes in the requirements might imply significant modifications. In this paper we show a solution based on answer set programming, which is a logic-based knowledge representation formalism well suited for a compact description of (re)configuration problems. Its applicability is demonstrated on simple abstractions of several real-world scenarios. The evaluation of our solution on a set of benchmark instances derived from commercial (re)configuration problems shows practical applicability.

1 Introduction

Reconfiguration is an important task in the after-sale life-cycle of configurable products and services, because requirements for these products and services are changing in parallel with the customers' business [6; 2]. In order to keep a product or a service up-to-date a re-engineering organization has to decide which modifications should be introduced to an existing configuration such that the new requirements are satisfied but change costs are minimized.

Following the knowledge based configuration approach, we formulate reconfiguration problem instances as extensions of declaratively defined configuration problem instances where configurations are represented by facts and requirements are expressed by logical descriptions. These requirements may be partitioned into customer requirements and system specific configuration requirements. A configuration is simply defined as a subset of a logical model of the requirements. Informally, a reconfiguration problem instance is generated by an adaption of the requirements resulting in a new set of requirements and therefore a new instance of a configuration problem is formulated. Subsequently, given legacy configurations have to be adapted to configurations for the new requirements. In our approach, the knowledge base comprises two parts, the description of the new configuration problem instance and transformation knowledge regarding reuse and deletion of parts of a legacy configuration. The first part is a usual instance of a configuration problem where all valid configurations are specified by the set of adapted requirements. The second part describes a mapping from the pieces of the legacy configuration to the ontology of the new configuration problem instance. Technically speaking this is a mapping from facts describing the legacy configuration to facts in the ontology of the new configuration problem instance. For generating a reconfiguration the problem solver has to decide which parts of the legacy configuration are either reused or deleted and which new parts have to be created.

We introduce general definitions for (re)configuration problems employing Herbrand-models of logical descriptions. Based on these definitions it is simple to see that configuration and reconfiguration problems fall into the same complexity classes. Because of the remarkable advances of answer set programming (ASP) [8; 5] we base our implementation on this reasoning framework. ASP was first applied to configuration problems by [9]. In particular, we provide modeling patterns for configuration and reconfiguration which allow the generation of optimized reconfigurations exploiting standard ASP solvers. Finally, our evaluation shows that the proposed method solves reconfiguration problem instances which are practically interesting for industrial applications.

In Section 2 we present an introductory example of a configuration problem and some reconfiguration scenarios. Then, configuration problems are defined in Section 3. In Section 4 a review of the basic concepts of ASP is given followed by an exemplification of modeling in Section 5. Section 6 provides the definition of reconfiguration problems. Subsequently, modeling patters and an example of their application are provided in Section 7. Finally, we show the results of an evaluation in Section 8 and conclude in Section 9.

2 Example

Let us exemplify different configuration and reconfiguration scenarios on a problem which is a simple abstraction of several configuration problems occurring in practice, i.e. entities may be contained in other entities but some restrictions must be fulfilled. We employ the ontology comprising the concepts *person*, *thing*, *cabinet*, and *room* where persons are related to things, things are related to cabinets, cabinets are related to

*This work has been developed within the scope of the project RECONCILE (reconciling legacy instances with changed ontologies) and was funded by FFG FIT-IT (grant number 825071).

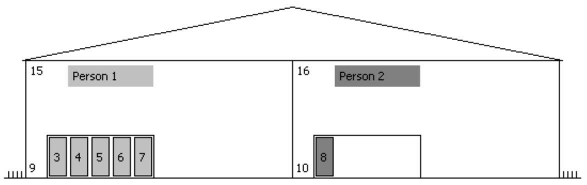

Figure 1: Solution of the sample house configuration problem. The house configuration includes rooms 15 and 16, two cabinets 9 and 10, and six things numbered from 3 to 8

rooms, and rooms are related to persons. These relations are modeled either by roles, associations, or predicate symbols depending on the modeling language (e.g. description logic, UML, or predicate logic).

As input to the configuration problem an ownership relation between persons and things is provided. We call this input a customer requirement since it reflects the individual needs of a customer using a configuration system whereas configuration requirements specify the properties of the system to be configured. Each person can own any number of things but each thing belongs to only one person. The problem is to place these things into cabinets and the cabinets into rooms of a house such that the following configuration requirements are fulfilled:

- each thing must be stored in a cabinet;
- a cabinet can contain at most 5 things;
- every cabinet must be placed in a room;
- a room can contain at most 4 cabinets;
- a person can own any number of rooms;
- each room belongs to a person;
- and a room may only contain cabinets storing things of the owner of the room.

In order to keep the example simple we only consider configuration of one house and represent all individuals using unique integer identifiers.

Informally, a configuration is every instantiation of the relations which satisfies all requirements.

Let a sample house problem instance include two persons such that the first person owns five things numbered 3 to 7 and the second person owns one thing 8. A solution for this house configuration problem instance is shown in Figure 1.

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

- a cabinet is either small or high;
- a long thing can only be put into a high cabinet;
- a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room;
- all legacy cabinets are small.

The customer requirements, in this case, define for each thing if it is long or short. For instance, the customer provides information that the things 3 and 8 are long; all others are short.

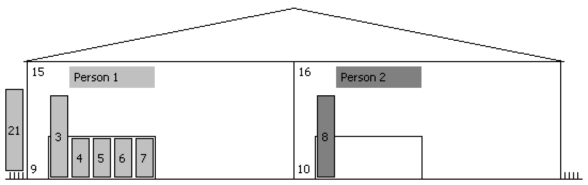

Figure 2: Reconfiguration initial state

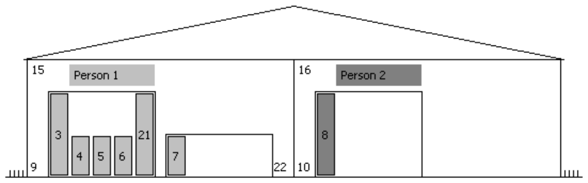

Figure 3: Reconfiguration solution 1

Moreover, the first person gets an additional long thing 21. The changes to the legacy configuration are summarized in Figure 2 showing an inconsistent configuration, where thing 21 is not placed in any of the cabinets, and cabinets 9 and 10 are too small for things 3 and 8.

To obtain a solution which is shown in Figure 3 the reconfiguration process changes the size of cabinets 9 and 10 to high and puts the new thing 21 into cabinet 9. A new small cabinet 22 is created for thing 7.

In our reconfiguration process every modification to the existing configuration, i.e. reusing/deleting/creating individuals and their relations, is associated with some cost. Therefore the reconfiguration problem is to find a consistent configuration by removing the inconsistencies and minimizing the costs involved. Different solutions will be found depending on the given modification costs. If, for example, the costs for adding a new high cabinet are less than the cost for changing an existing small cabinet into a high cabinet, then the previous solution should be rejected as its costs are too high. One of the solutions with less reconfiguration costs (see Figure 4) includes two new cabinets 22 and 23, because this is cheaper than converting the existing small cabinets into high cabinets. Also it contains the empty cabinet 10 because it's cheaper to keep the cabinet than to delete it. Note, this behavior can be controlled by the domain specific costs.

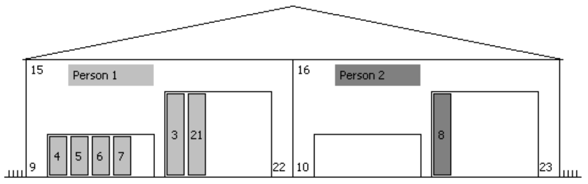

Figure 4: Reconfiguration solution 2

3 Configuration problems

We employ a definition of configuration problems based on logical descriptions [9; 3]. The basic idea is that every finite Herbrand-model contains the description of exactly one configuration.

The description of a configuration is defined by relations expressed by a set of predicates $\mathbf{P_S}$. This set of predicates is called the *solution schema*. For our example the solution schema consists of the four unary predicates `thing/1`, `person/1`, `cabinet/1` and `room/1` representing the individuals and the four binary predicates, namely `personTOthing/2`, `personTOroom/2`, `roomTOcabinet/2` and `cabinetTOthing/2` representing the relations. An instantiation of this solution schema corresponds to a configuration. A fragment of this instantiation is presented below.

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

Note, this description of a configuration generalizes the component/port models or variable/value based descriptions of a configuration.

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

Definition 1 (Instances of configuration problems) *A configuration problem instance* $\langle \mathbf{REQ}, \mathbf{P_S} \rangle$ *is defined by a set of logical sentences* $\mathbf{REQ}$ *representing requirements and* $\mathbf{P_S}$ *a set of predicate symbols representing the solution schema. For optimization purposes an objective function* $f(\mathbf{S}) \mapsto \mathbb{N}$ *maps any set of atoms* $\mathbf{S}$ *to positive integers where* $\mathbf{S}$ *contains only atoms whose predicate symbols are in* $\mathbf{P_S}$.

Let $\mathcal{HM}(\mathbf{L})$ denote the set of Herbrand-models of a set of logical sentences $\mathbf{L}$ for a given semantics.

Definition 2 (Configuration) $\mathbf{S}$ *is a configuration for a configuration problem instance* $\mathrm{CPI} = \langle \mathbf{REQ}, \mathbf{P_S} \rangle$ *iff there is a Herbrand-model* $\mathbf{M} \in \mathcal{HM}(\mathbf{REQ})$ *and* $\mathbf{S}$ *is the set of* all *the elements of* $\mathbf{M}$ *whose predicate symbols are in* $\mathbf{P_S}$ *and* $\mathbf{S}$ *is finite, i.e.* $\mathbf{S} = \{\mathrm{p}(\overline{\mathrm{t}}) | \mathrm{p} \in \mathbf{P_S} \text{ and } \mathrm{p}(\overline{\mathrm{t}}) \in \mathbf{M})\}$. *By* $\mathrm{p}(\overline{\mathrm{t}})$ *we denote a ground instance of* p *with a term vector* $\overline{\mathrm{t}}$.

$\mathbf{S}$ *is an optimal configuration for* CPI *iff* $\mathbf{S}$ *is a configuration for* CPI *and there is no configuration* $\mathbf{S}'$ *of* CPI *s.t.* $f(\mathbf{S}') < f(\mathbf{S})$.

Definition 3 (Configuration problems) *Let the instances of configuration problems be defined by* $\langle \mathbf{REQ}, \mathbf{P_S} \rangle$ *and objective functions* $f(\cdot)$.

Decision problem: *Given a set of atoms* $\mathbf{S}$. *Decide if* $\mathbf{S}$ *is a configuration for a configuration problem instance.*

Generation (optimization) problem: *Generate a set of atoms* $\mathbf{S}$ *s.t.* $\mathbf{S}$ *is a configuration (an optimal configuration) for a configuration problem instance.*

The set of Herbrand-models depends on the semantics of the employed logic. In this paper, we apply answer set programming and a stable model semantics for knowledge representation and reasoning because this approach allows a concise and modular specification, assures decidability, and avoids the inclusion of unjustified atoms (e.g. unjustified components) in configurations [9].

4 Overview on answer set programming

ASP is based on a decidable fragment of first-order logic enhanced with default negation and aggregation. We give a brief summary of the employed ASP variant and language constructs as needed. A detailed discussion of ASP can be found in [5; 4].

We start our introduction with rules without variables and introduce logical variables afterwards. A rule has the structure $C_0 \leftarrow C_1, \ldots, C_n$. Elements $C_1, \ldots, C_n$ on the right-hand-side (the body of a rule) are either literals or *weight* constraints. A literal is either an atom or a default negated atom. Default negation is expressed by *not*. C_0 (head of the rule) is either an atom or a weight constraint. We do not consider default negation on the left-hand-side and in weight constraints. If all C_i are literals then such a rule is called a *normal* rule.

To be able to express the requirements of our example domain we introduce a simplified version of weight constraints and their special case – *cardinality* constraints [9; 8; 4]. Weight constraints are of the form $l \leq \{a_1 = w_1, \ldots, a_n = w_m\} \leq u$ where a_i are atoms, w_j are integers representing weights of corresponding atoms and l, u are integers specifying lower and upper bounds. Given a set of atoms $\mathbf{M}$ representing a Herbrand-interpretation, the interpretation of a weight constraints evaluates to *true* iff the sum of weights of literals $a_1, \ldots, a_n$ which are contained in $\mathbf{M}$ is between l and u. E.g. $0 \leq \{a = 1, b = 2\} \leq 2$ is satisfied by $\emptyset$, $\{a\}$ or $\{b\}$ but not by $\{a, b\}$. Missing lower or upper bounds express the fact that there are no limits. Cardinality constraints are of the form $l \leq \{a_1, \ldots, a_n\} \leq u$ where each weight is considered to be equal 1. As usually (negated) atoms in the body of the rule are *true* if they are (not) in $\mathbf{M}$.

The semantics of a set of rules is defined by a stable model semantics. We give a brief informal description of this semantics for the restricted version employed in this paper and refer the reader to [8] for an in-depth exposition. A set of ground atoms $\mathbf{M}$ is a stable model for a set of rules $\mathbf{RU}$ iff two properties are fullfiled: (1) $\mathbf{M}$ *satisfies* all rules in $\mathbf{RU}$ and (2) every atom in $\mathbf{M}$ is *justified* by a reduced rule set $\mathbf{RU}^{\mathbf{M}}$. A rule is satisfied by a set of ground atoms $\mathbf{M}$ iff $\mathbf{M}$ satisfies C_0 or there exists a literal $C_1, \ldots, C_n$ which is not satisfied by $\mathbf{M}$. An empty body of a rule is always satisfied. A rule with empty head is satisfied iff one literal in the body is not satisfied. The precise semantics of justification is expressed by a reduction of the rule set $\mathbf{RU}$. Given $\mathbf{RU}$ and depending on the set of atoms $\mathbf{M}$, the reduct $\mathbf{RU}^{\mathbf{M}}$ is generated as follows. In our simplified version, default negated atoms are replaced in the rules $\mathbf{RU}$ according to their truth value w.r.t. $\mathbf{M}$, i.e. *not* a is *true* iff $a \notin \mathbf{M}$. Rules in $\mathbf{RU}$ are deleted if the head does not include an atom of $\mathbf{M}$ or some of the upper bounds are violated. Note, weight constraints in the head of a rule may comprise several atoms. Roughly speaking an atom in **M** is justified iff it is contained in the head of a rule and all atoms and weight constraints of this rule are justified. *True* is always and *false* is never justified. A weight/cardinality constraint in the body of a rule is justified if enough atoms contained in the weight/cardinality constraint are justified s.t. the lower bound is met. Facts are rules with *true* as body. Justifications must be acyclic. For instance, $0 \leq \{a, b\} \leq 1 \leftarrow c$ is satisfied by $\{a\}$ but $\{a\}$ is not justified. However, if we add the fact c to the knowledge base, $\{c\}$, $\{c, a\}$, and $\{c, b\}$ are stable models.

In order to allow logical variables and functional symbols but to guarantee decidability the set of allowed rules is restricted. Potassco [4] requires level-restricted programs. The basic idea is that for each variable V in a rule there is an unnegated atom a in the body s.t. the potentially derivable ground instances of a are limited. If such an atom is available the ground terms to which V needs to be instantiated are known a-priori. I.e. every variable in a rule must be bound to a finite set of ground terms via a predicate that is not subject to a positive recursion (recursion over unnegated atoms) through that rule.

For a succinct specification of facts in our example we use so-called intervals, e.g. $\mathtt{person(1..2).}$ corresponds to the facts $\mathtt{person(1).\ person(2).}$ To exemplify the application of cardinality constrains, let an ASP program contain the facts:

$$\mathtt{thing(3..4).}\quad \mathtt{cabinetDomain(9..10).}$$

In order to formulate weight constraints concisely, so called conditional literals are supported. The basic idea is that conditional literals serve as a generator for producing a set of atoms. The constraint

$$\mathtt{1\{cabinetTOthing(X,Y)\!: cabinetDomain(X)\}1} \leftarrow \mathtt{thing(Y).}$$

where $\mathtt{cabinetTOthing(X,Y) : cabinetDomain(X)}$ is a conditional literal, which is expanded to

$$\mathtt{1\{cabinetTOthing(9,3), cabinetTOthing(10,3)\}1} \leftarrow \mathtt{thing(3).}$$
$$\mathtt{1\{cabinetTOthing(9,4), cabinetTOthing(10,4)\}1} \leftarrow \mathtt{thing(4).}$$

expressing that things 3 and 4 must be connected to exactly one of the cabinets 9 and 10. Conditional literals can be used in weight constraints in place of atoms, where the *conditional part* (e.g. $\mathtt{cabinetDomain(X)}$) is a (conjunction of) *domain predicate*(s) preceded by the *main part*. As usual, strings starting with upper case letters are logical variables. The instantiation of domain predicates is defined by non-recursive normal rules and ground facts. For instantiating conditional literals we have to distinguish between *local* and *global* variables. A variable is local iff it appears only in a conditional literal, e.g. X is local in our example. All other variables are global, e.g. Y. During grounding of the rules, global variables are instantiated first. Then the main part of the conditional literal is expanded for the instantiations of the local variables where the conditional part is fulfilled.

Note, in Potassco [4] weight constraints are declared by square brackets $\mathtt{l} \leq [\mathtt{L_1} = \mathtt{w_1}, \ldots, \mathtt{L_n} = \mathtt{w_n}] \leq \mathtt{u}$, where $\mathtt{L_i}$ is a literal and $\mathtt{w_i}$ is a numerical value representing its weight. Literals $\mathtt{L_i}$ could be equal. Curly brackets are employed to define cardinality constraints where duplicated literals are removed.

Answer set programming solvers like [8; 5; 4] offer optimization services. In particular, the statement $\#\mathtt{minimize}[\mathtt{L_1} = \mathtt{w_1}@\mathtt{p_1}, \ldots, \mathtt{L_n} = \mathtt{w_n}@\mathtt{p_n}].$ allows minimization. The minimization statement is similar to the weight constraints with a possibility to assign a priority level $\mathtt{p_i}$ to each weighted literal. Instead of $\#\mathtt{minimize}$ also $\#\mathtt{maximize}$ could be used. An answer set is optimal iff the sum of the weights of literals which are satisfied in this answer set is minimal (maximal) among all answer sets of a given program. Optimization is performed in the order of priorities starting from the highest priority value.

5 Defining configuration problem instances

In [9] various modeling patterns based on weight constraints were introduced. A fixed set of ground facts define the individuals which are employed for a configuration. This fixed set of ground facts in conjunction with the level-restriction place an upper bound on the size of the number of grounded rules and therefore decidability is guaranteed. At the current state of research such an upper bound on the number of individuals is necessary for many applications. In particular, it is well known from database theory that so called tuple generating dependencies lead to undecidability even under rather strict syntactical restrictions [1]. A tuple generating dependency is $\forall\overline{X}\forall\overline{Y}\phi(\overline{X},\overline{Y}) \rightarrow \exists\overline{Z}\psi(\overline{X},\overline{Z})$ where $\phi(\overline{X},\overline{Y})$ and $\psi(\overline{X},\overline{Z})$ are conjunctions of atoms and $\overline{X}, \overline{Y}$, and $\overline{Z}$ are representing vectors of logical variables. Unfortunately, such rules may occur in configuration problem instances. E.g. if a condition holds, a specific individual of some type must exist and this individual must be connected to some other individuals.

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

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

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

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

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

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

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

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

```
cabinetDomain(9..14).
2{cabinet(X):cabinetDomain(X)}6.
cabinet(X) :- cabinetDomain(X), cabinetDomain(Y),
                                cabinet(Y), X<Y.
```

Cardinality restrictions given in Section 2 are encoded with cardinality constraints, where one direction of an association is encoded as a generation rule (see Section 4) and the other direction as a constraint. Such encoding corresponds to Guess/Check/Optimize pattern [5]. Note, the cardinality constraints just as the weight constraints require that logical variables appear in domain predicates. Therefore, we have to use `pDomain` predicates rather than `p` predicates, e.g. `cabinetDomain(X)` instead of `cabinet(X)`. However, individuals employed in relations must also be contained in the corresponding types (see the last four rules of the next sequence of rules). By these rules we avoid situations where an individual is used in a relation but not included in the bounded type. In our example, if the program asserts `cabinetTOthing(14,1)` then `cabinet(14)` is also asserted.

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

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

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

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

$$\mathtt{cost}(\mathtt{create}(\mathrm{p}(\overline{\mathtt{X}})),\mathtt{W}) \leftarrow \mathrm{p}(\overline{\mathtt{X}}), \alpha(\overline{\mathtt{X}},\overline{\mathtt{Y}},\mathtt{W}).$$

such that for each atom $\mathrm{p}(\overline{\mathtt{t}})$ in $\mathbf{S}$ the answer set contains an atom $\mathtt{cost}(\mathtt{create}(\mathrm{p}(\overline{\mathtt{t}}),\mathtt{w}))$ where $\mathtt{w}$ is an integer. E.g.:

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

All other creation costs are expressed in the same way. We minimize the sum of all costs by means of the following optimization statement:

```
#minimize[cost(X,W)=W@1].
```

For the given example the solver finds the optimal configuration including two cabinets and two rooms with the overall cost 40 (depicted in Figure 1).

```
{cabinet(10), cabinet(9), room(16), room(15), ...,
roomTOcabinet(15,9), roomTOcabinet(16,10),
cabinetTOthing(10,8) cabinetTOthing(9,7), ...,
cabinetTOthing(9,3), personTOroom(1,15),
personTOroom(2,16)}
```

6 Reconfiguration problems

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

Definition 4 (Instances of reconfiguration problems) *A reconfiguration problem instance* $\langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle, \mathbf{S}, \mathbf{T}\rangle$ *is defined by:* $\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle$ *an instance of a configuration problem,* $\mathbf{S}$ *a legacy configuration and* $\mathbf{T}$ *a set of logical sentences representing the transformation constraints regarding the legacy configuration.*

For optimization purposes an objective function $g(\mathbf{S}, \mathbf{R}) \mapsto \mathbb{N}$ *maps legacy configurations* $\mathbf{S}$ *and configurations* $\mathbf{R}$ *of* $\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle$ *to positive integers.*

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

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

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

Definition 5 (Reconfiguration) $\mathbf{R}$ *is a reconfiguration for a reconfiguration problem instance* RCI $= \langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle,$ $\mathbf{S}, \mathbf{T}\rangle$ *iff there is a Herbrand-model* $\mathbf{M} \in \mathcal{HM}(\mathbf{REQ_R} \cup$

$\mathbf{S} \cup \mathbf{T}$) *and* $\mathbf{R}$ *is the set of all the elements of* $\mathbf{M}$ *whose predicate symbols are in* $\mathbf{P_R}$ *and* $\mathbf{R}$ *is finite.*

$\mathbf{R}$ *is an optimal reconfiguration for* RCI *iff* $\mathbf{R}$ *is a reconfiguration for* RCI *and there is no reconfiguration* $\mathbf{R}'$ *of* RCI *s.t.* $g(\mathbf{S}, \mathbf{R}') < g(\mathbf{S}, \mathbf{R})$.

Reconfiguration problems are formulated analog to configuration problems.

Definition 6 (Reconfiguration problems) *The instances of reconfiguration problems are defined by a tuple* $\langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle, \mathbf{S}, \mathbf{T}\rangle$ *and objective functions* $g(\cdot,\cdot)$.

Decision problem: *Given a set of atoms* $\mathbf{R}$. *Decide if* $\mathbf{R}$ *is a reconfiguration for a reconfiguration problem instance.*

Generation (optimization) problem: *Generate a set of atoms* $\mathbf{R}$ *s.t.* $\mathbf{R}$ *is a reconfiguration (an optimal reconfiguration) for a reconfiguration problem instance.*

Because we can reduce configuration problems to reconfiguration problems and vice versa the following property follows trivially.

Property 1 *Employing a logical representation language for representing instances of configuration problems and reconfiguration problems whose satisfiability problem is at least NP-complete, generating a(n optimal) reconfiguration is as hard as generating a(n optimal) configuration w.r.t. computational complexity.*

7 Defining reconfiguration problem instances

In the following we show typical formalization patterns and apply them to our example. The set of atoms $\{\mathtt{legacyConfig}(a) | a \in \mathbf{S}\}$ describes the atoms of the legacy configuration $\mathbf{S}$. Note, the definition of reconfiguration problems does not employ first-order logic constructs in order to avoid unnecessary restrictions. However, to facilitate a concise description of the problem we introduce the predicate `legacyConfig/1` to allow quantification over the elements of the legacy configuration. Note, we could rewrite all shown axioms to propositional logic.

For the transformation sentences $\mathbf{T}$ we employ the following general patterns. For reusing parts of the legacy configuration the problem solver has to make the decision either to *reuse* or to *delete*. This is expressed by `reuse(a)` and `delete(a)` atoms where a is an element of $\mathbf{S}$. For each atom a in $\mathbf{S}$ either `reuse(a)` or `delete(a)` must hold. Based on these atoms additional configuration constraints can be defined which describe the proper reuse or deletion of a part of the legacy configuration represented by atom a. In our case, reusing an atom a of the legacy configuration implies the assertion of this atom, whereas deletion requires that the atom is not asserted. In addition, costs are associated to each `reuse(a)` or `delete(a)` operation. This is expressed by the atom `cost(reuse(a), w)` or `cost(delete(a), w)` where `a` is an element of $\mathbf{S}$ and `w` is an integer specifying the corresponding costs. Furthermore, we require that in each model which contains `reuse(a)` or `delete(a)` also `cost(reuse(a), w)` or `cost(delete(a), w)` is contained in order to have defined reuse or deletion costs. The conjunctions $\beta(\overline{X}, \overline{Y}, W)$ and $\gamma(\overline{X}, \overline{Y}, W)$ are employed to define case specific costs.

For each $\mathtt{p} \in \mathbf{P_S}$ include the following axioms in $\mathbf{T}$:

$\mathtt{1\{reuse(p(\overline{X})), delete(p(\overline{X}))\}1} \leftarrow \mathtt{legacyConfig(p(\overline{X}))}.$

$\mathtt{p(\overline{X})} \leftarrow \mathtt{reuse(p(\overline{X}))}.$

$\leftarrow \mathtt{p(\overline{X}), delete(p(\overline{X}))}.$

$\mathtt{cost(reuse(p(\overline{X})), W)} \leftarrow \mathtt{reuse(p(\overline{X}))}, \beta(\mathtt{\overline{X}, \overline{Y}, W}).$

$\mathtt{cost(delete(p(\overline{X})), W)} \leftarrow \mathtt{delete(p(\overline{X}))}, \gamma(\mathtt{\overline{X}, \overline{Y}, W}).$

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

$\mathtt{pDomain(X)} \leftarrow \mathtt{legacyConfig(q(\ldots, X, \ldots))}.$

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

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

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

However, the two other rules for bounded types are changed. In particular, we have to adapt the symmetry breaking pattern of configurations. The reason is that there are two different types of individuals contained in `pDomain`, those which are reused and those which are newly generated. Symmetry breaking does not apply to the reused individuals because they may be linked to other reused individuals. Therefore, exchanging these individuals potentially leads to different configurations. However, the newly generated individuals are interchangeable. We describe them by `pDomainNew/1` for the bounded type `p`. We use `pNewOffset` to generate new identifiers. I.e. the pattern is

$\mathtt{pDomainNew(pNewOffset + 1\ ..\ pNewOffset + pUpper)}.$

$\mathtt{pDomain(X)} \leftarrow \mathtt{pDomainNew(X)}.$

$\mathtt{p(X)} \leftarrow \mathtt{pDomainNew(X), pDomainNew(Y), p(Y), X < Y}.$

In our example, the reconfiguration problem consists of additional customer and configuration requirements described in Section 2. The solution schema for the reconfiguration problem is an extension of the solution schema of the original configuration problem by `cabinetHigh/1`, `cabinetSmall/1`, `thingLong/1` and `thingShort/1` predicates. The additional requirements of the customer are expressed by:

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

The legacy configuration presented in Section 3 is encoded using `legacyConfig` predicate as described above.

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

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

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

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

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

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

```
cabinetDomainNew(22..28).
cabinetDomain(X) :- cabinetDomainNew(X).
2{cabinet(X):cabinetDomain(X)}7.
cabinet(X) :- cabinetDomainNew(X), cabinet(Y), X<Y,
            cabinetDomainNew(Y).
```

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

The transformation rules are implemented as described above. E.g.

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

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

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

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

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

Rules for deducing the costs of reuse and deletion are formulated as described above.

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

Modification of the costs results into different optimal reconfigurations. Let us assume the sales-department changes both the costs of deletion of a cabinet and the costs of increasing the height of a cabinet to 10, and decreases the creation costs of new high and small cabinets to 2 and 1 respectively. In this case the solutions returned by a solver will include the one presented in Figure 4. Given their simplicity, the presented optimal solutions were found in milliseconds.

8 Evaluation

The evaluation of our approach was done on a set of test cases derived from four reconfiguration scenarios encountered by us in practice. Each scenario can be represented as an instance of the (re)configuration problem presented in Section 2. In the *empty* reconfiguration scenario the legacy configuration is empty and the customer requirements contain sets of things and persons owning 5 things each. Every thing is labeled as short. The reconfiguration process should create missing cabinets, rooms as well as all required relations.

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

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

The last scenario, *swap*, describes a situation when the customer requirements include only one person, who owns 35 things. In the legacy configuration the things are placed in 3 cabinets in the first room and in 4 cabinets in the second room. Moreover, one of the things in the second room is labeled as high in the customer requirements. Given the costs schema presented above, the solution corresponds to a rearrangement of the cabinets in the rooms such that a high cabinet can be placed in one of these rooms. All these scenarios can be easily scaled by increasing the number of things. The number of persons in the empty, long and new room scenarios can always be computed given the number of things.

Experiments were performed using Potassco 3.0.3 on Core2 Duo 3Ghz with 4Gb RAM. In our experiments we con-

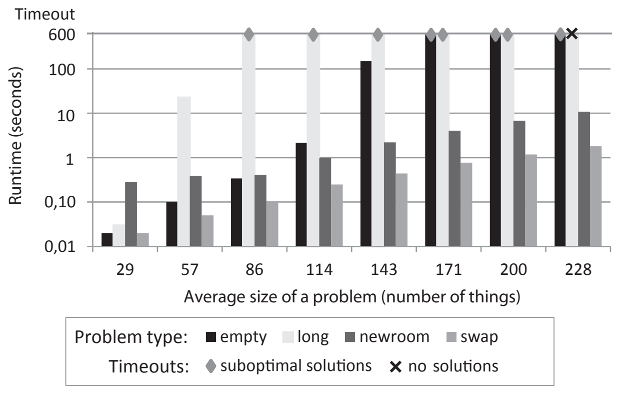

Figure 5: Evaluation results

sidered only creation costs for newly generated cabinets and rooms because these are the dominant costs for our application domain. The performance of the reconfiguration process is presented in Figure 5. Potassco was able to find optimal solutions within 600 seconds for all instances of the new room and swap scenarios. Optimal solutions were also found for small and mid-size instances of the empty scenario. For all other instances at least one suboptimal solution was found. The long scenario included the hardest problems. The solver did not find any solutions for one of them in 600 seconds. This was the only unsolved problem instance in the whole experiment. Because the solved instances are comparable to real world applications based on our experiences, we consider the proposed reconfiguration method as feasible for a practically interesting set of reconfiguration problem instances.

9 Conclusions and related work

The existing approaches for reconfiguration can be separated into revision-based [7; 10] and model-based [11]. The revision-based approaches employ a knowledge base describing "fixes", i.e. reconfiguration operations and configuration invariants [7]. A solution requires that there is a *sequence* of operations which transform the legacy configuration into a new configuration. The approach of [11] views reconfiguration as a consistency-maintenance (diagnosis) problem, where a solution corresponds to a consistent set of assumptions s.t. requirements are implied. Similarly, our approach can be seen as searching for a consistent (optimal) set of assumptions regarding reuse or deletion of parts of the legacy configuration and creation of new parts. This search is provided by the ASP reasoning system, implementing a *correct and complete* problem solving method. No additional diagnosis component is required. Regarding the revision-based approach, our domains do not need the computation of sequences of operations, because if a reconfiguration is found, a sequence of real-world change operations can be easily derived. Thus, we can avoid the additional combinatorial explosion introduced by permutations of change operations. However, we can view our approach as a form of the revision-based method assuming that all change operations are executed simultaneously. The effects of these operations and the combination of allowed operations are described by the transformation knowledge. Thus we can model complex "fix" operations which involve the reuse of several parts of the legacy configuration and which have multiple effects such as creating new parts or deleting existing ones.

To sum up, we have developed a method which allows the modeling of reconfiguration problems based on legacy configurations, transformation knowledge, and a new configuration problem instance. We showed various modeling patterns and implemented the approach based on ASP. Evaluation results show the feasibility for practical applications.

References

[1] A. Calì, G. Gottlob, and M. Kifer. Taming the infinite chase: Query answering under expressive relational constraints. In *11th International Conference on Principles of Knowledge Representation and Reasoning*, pages 70–80. AAAI Press, 2008.

[2] A. Falkner and A. Haselböck. Challenges of Knowledge Evolution in Practice. In *Workshop on Intelligent Engineering Techniques for Knowledge Bases (IKBET 2010)*, pages 1–5, 2010.

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

[4] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to gringo, clasp, clingo and iclingo, 2010.

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

[6] P. Manhart. Reconfiguration - A problem in search of solutions. In D. Jannach and A. Felfernig, editors, *IJCAI'05 Configuration Workshop*, pages 64–67, 2005.

[7] T. Männistö, T. Soininen, J. Tiihonen, and R. Sulonen. Framework and conceptual model for reconfiguration. In B. Faltings, E. C. Freuder, and G. Friedrich, editors, *AAAI'99 Workshop on Configuration*, volume 99, pages 59–64, 1999.

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

[9] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge*, pages 195–201, 2001.

[10] L. Stojanovic, A. Maedche, N. Stojanovic, and R. Studer. Ontology evolution as reconfiguration-design problem solving. In *2nd International Conference on Knowledge Capture*, pages 162—171, New York, NY, USA, 2003. ACM Press.

[11] M. Stumptner and F. Wotawa. Model-based reconfiguration. In J. S. Gero and F. Sudweeks, editors, *5th International Conference on Artificial Intelligence in Design*, pages 45–64, 1998.