(Re)configuration based on model generation[*]

Gerhard Friedrich
Anna Ryabokon
Alpen-Adria Universität, Klagenfurt, Austria
`firstname.lastname@aau.at`

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

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 its 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 [5, 1]. 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: (1) the description of the new configuration problem instance, where all valid configurations are specified by the set of adapted requirements and (2) transformation knowledge regarding reuse and deletion of parts of a legacy configuration. 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. Because of the remarkable advances of answer set programming (ASP) [7, 4, 3] we base our implementation on this reasoning framework. ASP was first applied to configuration problems by [8]. In particular, we provide modeling patterns for (re)configuration 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.

[*]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).

© G. Friedrich, A. Ryabokon, A.A. Falkner,
A. Haselböck, G. Schenner, H. Schreiner
This work is licensed under the
Creative Commons Attribution License.

2 Example

Let us exemplify different (re)configuration 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 rooms, and rooms are related to persons.

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: (1) each thing must be stored in a cabinet; (2) a cabinet can contain at most 5 things; (3) every cabinet must be placed in a room; (4) a room can contain at most 4 cabinets; (5) a person can own any number of rooms; (6) each room belongs to a person; and (7) a room may only contain cabinets storing things of the owner of the room.

In our simple example 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. The first person owns five things numbered 3 to 7 and the second person owns one thing 8. A solution for this problem instance is shown in Figure 1, including rooms 15 and 16 with cabinets 9 and 10. Reconfiguration is necessary, whenever the customer requirements or configuration requirements are changed. For instance, it must be differentiated between long and short things with the following additional requirements: (8) a cabinet is either small or high; (9) a long thing can only be put into a high cabinet; (10) a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room; and (11) all legacy cabinets are small.

The customer requirements, in this case, define for each thing if it is long or short. For instance, things 3 and 8 are long; all others are short. Moreover, the first person gets an additional long thing 21. The changes to the legacy configuration are summarized in Figure 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. Furthermore, cabinet 10 is not removed because it's cheaper to keep the cabinet than to delete it.

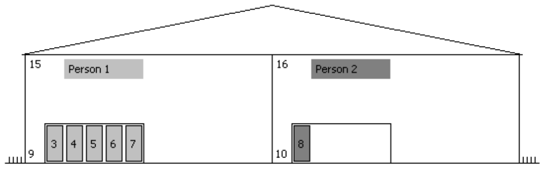

Figure 1: Initial configuration

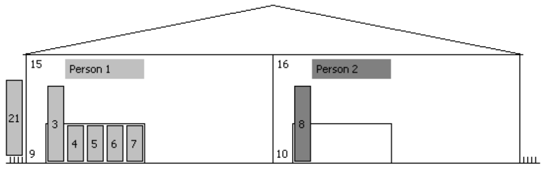

Figure 2: Reconfiguration – initial state

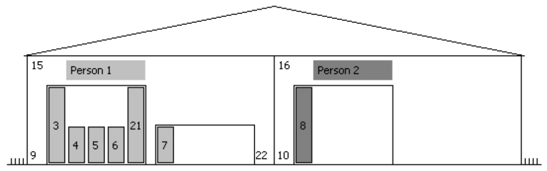

Figure 3: Reconfiguration solution 1

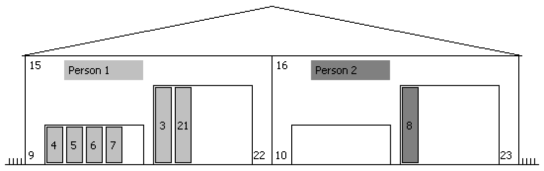

Figure 4: Reconfiguration solution 2

3 Short overview of answer set programming

Before we introduce (re)configuration problems we give a brief overview of ASP. ASP is based on a decidable fragment of first-order logic enhanced with default negation and aggregation. A detailed discussion of ASP can be found in [4, 3]. In order to simplify our presentation we avoid default negation and employ a restricted form of aggregation called *cardinality* constraint [8, 7, 3]. Cardinality constraints are of the form $l\{a_1,\dots,a_n\}u$ where a_i are atoms and l,u are integers specifying lower and upper bounds. By using cardinality constraints in the consequent of a rule, choices can be expressed, i.e. from the atoms a_i at least l but at most u must be true. Furthermore, the answer set semantic restricts the set of classical Herbrand models. Only those Herbrand models are accepted where every atom is either a fact or justified by a non-circular application of rules. For instance, $0\{a,b\}1 \leftarrow c$ 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, i.e. Herbrand models accepted by the answer set semantic of [7]. ASP allows logical variables under syntactical restrictions which assure the elimination of all variables by ground terms, i.e. during grounding of a knowledge-base a rule containing variables is replaced by a set of ground rules.

For a succinct specification of facts in our example we use so-called intervals, e.g. `person(1..2).` corresponds to the facts `person(1). person(2).` In order to formulate cardinality 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. To exemplify the application of cardinality constrains and conditional literals, let an ASP program contain the facts: `thing(3..4). cabinetDomain(9..10).` 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, is expanded to

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

and expresses that things 3 and 4 must be connected to exactly one of the cabinets 9 and 10. Conditional literals can be used in cardinality constraints in place of atoms, where the *conditional part* (e.g. `cabinetDomain(X)`) is a (conjunction of) *domain predicate*(s) preceded by the *main part*. All predicates of atoms in the conditional part must be domain predicates. Roughly speaking, domain predicates are predicates whose ground instantiations are the same for all answer sets, i.e. the ground instantiations can be determined without searching. 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.

Answer set programming solvers like [7, 4, 3] offer optimization services. In particular, the literal $\mathtt{\#minimize[a_1 = w_1,\dots,a_n = w_n]}$ allows minimization, where $\mathtt{a_i}$ are atoms and $\mathtt{w_i}$ are associated weights. An answer set is optimal iff the sum of the weights of atoms which are satisfied in this answer set is minimal among all answer sets of a given program.

4 Configuration problems

We employ a definition of configuration problems based on logical descriptions [8, 2]. 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 $\mathbf{P_S}$, which is called *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 `personTOthing/2`, `personTOroom/2`, `roomTOcabinet/2` and `cabinetTOthing/2` representing the relations. An instantiation of this solution schema corresponds to a configuration.

We assume that predicate symbols have a unique arity. The set of Herbrand-models is specified by a set of logical sentences **REQ**, which usually comprises *customer* and *configuration requirements*. The latter reflect the set of all allowed configurations for an artifact, whereas the customer requirements may comprise facts and logical sentences specifying the individual needs of customers. In our example customer requirements are expressed by facts describing the relation between persons and things. Configuration requirements, like "each thing must be stored in a cabinet" are expressed by logical sentences.

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* **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* **S** *to positive integers where* **S** *contains only atoms whose predicate symbols are in* $\mathbf{P_S}$.

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

Definition 2 (Configuration) **S** *is a configuration for a configuration problem instance* $\mathrm{CPI} = \langle \mathbf{REQ}, \mathbf{P_S} \rangle$ *iff there is a Herbrand-model* $\mathbf{M} \in \mathscr{H}(\mathbf{REQ})$ *and* **S** *is the set of* all *the elements of* **M** *whose predicate symbols are in* $\mathbf{P_S}$ *and* **S** *is finite, i.e.* $\mathbf{S} = \{\mathtt{p}(\overline{\mathtt{t}}) | \mathtt{p} \in \mathbf{P_S} \text{ and } \mathtt{p}(\overline{\mathtt{t}}) \in \mathbf{M})\}$. *By* $\mathtt{p}(\overline{\mathtt{t}})$ *we denote a ground instance of* p *with a term vector* $\overline{\mathtt{t}}$. **S** *is an optimal configuration for* CPI *iff* **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 generation (optimization) problems) *Let the instances of configuration problems be defined by* $\langle \mathbf{REQ}, \mathbf{P_S} \rangle$ *and objective functions* $f(\cdot)$. *Generate a set of atoms* **S** *s.t.* **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 ASP because this framework allows a concise and modular specification, assures decidability, and avoids the inclusion of unjustified atoms (e.g. unjustified components) in configurations [8].

In [8] various modeling patterns based on ASP were introduced. A fixed set of ground facts defines the individuals which are employed for a configuration. However, in many cases it is undesirable to consider only a fixed number of individuals used in a configuration. Therefore, we apply the following modeling pattern. Let `pLower` and `pUpper` represent the upper and lower number of individuals of type `p`. Such a type is called *bounded*. We require each individual of a configuration, represented by its unique identifier, to be a member of exactly one bounded type. To each bounded type a domain `pDomain` is associated, representing the set of possible individuals of the bounded type. We employ numbers as identifiers, starting from some offset. For every bounded type `p` we add the following axioms:

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

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

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

For some bounded types, e.g. `person/1` and `thing/1` the bounds `pLower` and `pUpper` coincide because the exact number of individuals employed in any configuration is known. In this case the fixed set of `p` facts can be asserted without using the rules presented above. In our example the customer provides a set of requirements for a configuration including definitions of person and thing individuals as well as their relations: `person(1..2). thing(3..8). personTOthing(1,3). ... 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 3) and the other direction as a constraint. Note, all predicates of atoms in the conditional part of a conditional literal must be domain predicates. Therefore, we have to use `pDomain` predicates rather than `p` predicates, e.g. `cabinetDomain(X)` instead of `cabinet(X)` (see the first rule of the next sequence of rules). However, individuals employed in relations must also be contained in the corresponding types (see the last two rules of the next sequence of rules as example). By these rules we avoid situations where an individual is used in a relation but not included in the bounded type.

```
1{cabinetTOthing(X,Y):cabinetDomain(X)}1 :- thing(Y).
:- 6 {cabinetTOthing(X,Y):thing(Y)}, cabinet(X).
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 **S** some costs. This can be achieved with the following modeling pattern. By the atom `cost(create(a,w))`, where a is an element of **S** and `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. The variable vectors $\overline{\mathtt{X}},\overline{\mathtt{Y}}$ can be exploited to formulate arbitrary usage of logical variables in the conjunction of atoms. For each $\mathtt{p} \in \mathbf{P_S}$ include axioms of the following form in **REQ**: $\mathtt{cost(create(p(}\overline{\mathtt{X}}\mathtt{)),W)} \leftarrow \mathtt{p}(\overline{\mathtt{X}}), \alpha(\overline{\mathtt{X}},\overline{\mathtt{Y}},\mathtt{W})$ such that for each atom $\mathtt{p}(\overline{\mathtt{t}})$ in **S** the answer set contains an atom $\mathtt{cost(create(p(}\overline{\mathtt{t}}\mathtt{),w))}$ where `w` is an integer. For example:

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

All other creation costs are expressed in the same way. We minimize the sum of all costs by means of the optimization statement: `#minimize[cost(X,W) = W]`. By this statement all `W` of atoms `cost(X,W)` are summed up. Only those answer sets which minimize this summation are optimal solutions.

For the given example the solver finds the optimal configuration including two cabinets and two rooms (depicted in Figure 1): `{cabinet(10), cabinet(9), room(16), room(15), ..., roomTOcabinet(15,9), ..., cabinetTOthing(10,8), ..., personTOroom(2,16)}`

5 Reconfiguration problems

We view reconfiguration as a new configuration-generation problem where parts of a *legacy configuration* are possibly reused. The conditions under which some parts of the legacy configuration can be reused and what the consequences of a reuse are, is expressed by a set of logical sentences **T** which relate the legacy configuration **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 a reconfiguration depend on the elements in a reconfiguration and 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 analogously to configurations as a finite subset of Herbrand-models.

Definition 5 (Reconfiguration) $\mathbf{R}$ *is a reconfiguration for a reconfiguration problem instance* $\mathrm{RCI} = \langle\langle \mathbf{REQ_R}, \mathbf{P_R}\rangle, \mathbf{S}, \mathbf{T}\rangle$ *iff there is a Herbrand-model* $\mathbf{M} \in \mathscr{H}(\mathbf{REQ_R} \cup \mathbf{S} \cup \mathbf{T})$ *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 analogously to configuration problems.

Definition 6 (Reconfiguration generation (optimization) 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* $g(\cdot,\cdot)$. *Generate a set of atoms* $\mathbf{R}$ *s.t.* $\mathbf{R}$ *is a reconfiguration (an optimal reconfiguration) for a reconfiguration problem instance.*

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.

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 by any rule application of the knowledge base. 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 $\mathbf{T}$:

$$\mathtt{1\{reuse(p(\overline{X})),delete(p(\overline{X}))\}1} \leftarrow \mathtt{legacyConfig(p(\overline{X}))}.$$
$$\leftarrow \mathtt{p(\overline{X}),delete(p(\overline{X}))}. \quad \mathtt{p(\overline{X})} \leftarrow \mathtt{reuse(p(\overline{X}))}.$$
$$\mathtt{cost(reuse(p(\overline{X})),W)} \leftarrow \mathtt{reuse(p(\overline{X}))}, \beta(\mathtt{\overline{X},\overline{Y},W}).$$
$$\mathtt{cost(delete(p(\overline{X})),W)} \leftarrow \mathtt{delete(p(\overline{X}))}, \gamma(\mathtt{\overline{X},\overline{Y},W}).$$

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

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

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:

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

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:

$$\texttt{pDomainNew(pNewOffset} + 1\ ..\ \texttt{pNewOffset} + \texttt{pUpper)}. \qquad \texttt{pDomain(X)} \leftarrow \texttt{pDomainNew(X)}.$$
$$\texttt{p(X)} \leftarrow \texttt{pDomainNew(X),pDomainNew(Y),p(Y),X} < \texttt{Y}.$$

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 customer requirements 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 4 is encoded using legacyConfig predicate:

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

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 fourth rule of the following rule sequence contains a weighted constraint. To each atom roomTOcabinetSlot(X,Y,S) a weight S is associated. The weighted constraint is true if the sum of the weights of the true atoms are greater or equal to 5. In this case an inconsistency is detected.

```
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 domain of cabinets is extended with additional individuals that might be required in a new configuration. The number of new elements in the cabinet domain corresponds to the number of things in the modified problem. The upper number `pUpper` of cabinet individuals is set to 7, because 7 things must be stored in the house. The extension of the room domain is done in the same way.

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

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))` as well as for `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 the solver is able to find a set of optimal reconfigurations including the one presented in Figure 3.

Modification of the costs results in 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.

6 Evaluation

We are challenged by various real-world configuration problems from technical domains like telecommunication or railway safety systems with structure and complexity similar to the example used in this paper. The evaluation of our approach is performed on test cases[1] from such practical configuration scenarios. 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. Things are labeled as short. The reconfiguration process should create missing cabinets, rooms and 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.

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

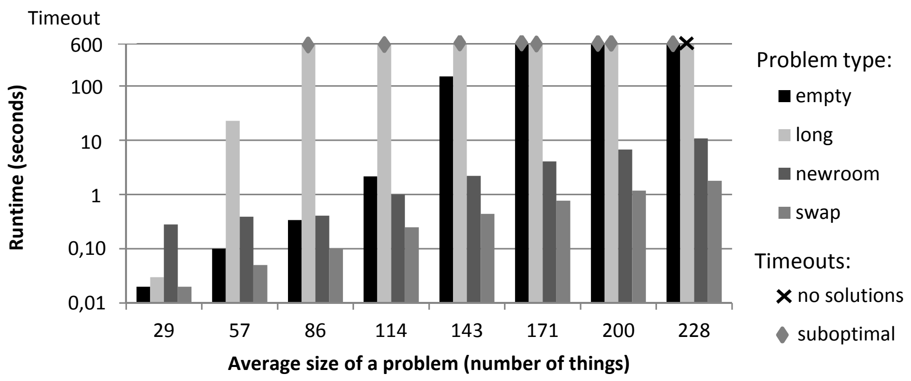

Figure 5: Evaluation results

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 long 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 considered 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.

7 Conclusions and related work

The existing approaches for reconfiguration can be separated into revision-based [6] and model-based [9]. The revision-based approaches employ a knowledge base describing “fixes”, i.e. reconfiguration operations and configuration invariants [6]. A solution requires that there is a *sequence* of operations which transform the legacy configuration into a new configuration. The approach of [9] 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] Andreas Falkner & Alois Haselböck (2010): *Challenges of Knowledge Evolution in Practice*. In: *Workshop on Intelligent Engineering Techniques for Knowledge Bases (IKBET 2010)*, pp. 1–5.

[2] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach & Markus Stumptner (2004): *Consistency-based diagnosis of configuration knowledge bases*. *Artificial Intelligence* 152(2), pp. 213–234, doi:10.1016/S0004-3702(03)00117-6.

[3] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub & Sven Thiele (2010): *A user's guide to gringo, clasp, clingo and iclingo*. Available at `http://potassco.sourceforge.net/`.

[4] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri & Francesco Scarcello (2006): *The DLV system for knowledge representation and reasoning*. *ACM Transactions on Computational Logic (TOCL)* 7(3), pp. 499–562, doi:10.1145/1149114.1149117.

[5] Peter Manhart (2005): *Reconfiguration - A problem in search of solutions*. In Dietmar Jannach & Alexander Felfernig, editors: *IJCAI'05 Configuration Workshop*, pp. 64–67.

[6] Tomi Männistö, Timo Soininen, Juha Tiihonen & Reijo Sulonen (1999): *Framework and conceptual model for reconfiguration*. In Boi Faltings, Eugence C. Freuder & Gerhard Friedrich, editors: *AAAI'99 Workshop on Configuration*, 99, pp. 59–64.

[7] Patrik Simons, Ilkka Niemelä & Timo Soininen (2002): *Extending and implementing the stable model semantics*. *Artificial Intelligence* 138(1-2), pp. 181–234, doi:10.1016/S0004-3702(02)00187-X.

[8] Timo Soininen, Ilkka Niemelä, Juha Tiihonen & Reijo Sulonen (2001): *Representing configuration knowledge with weight constraint rules*. In: *1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge*, pp. 195–201.

[9] Markus Stumptner & Franz Wotawa (1998): *Model-based reconfiguration*. In John S. Gero & Fay Sudweeks, editors: *5th International Conference on Artificial Intelligence in Design*, pp. 45–64.