LoCo — A Logic for Configuration Problems[1]

Markus Aschinger[2] and **Conrad Drescher**[2] and **Heribert Vollmer**[3]

Abstract. LoCo is a fragment of classical first order logic tailored for expressing 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. Computing configurations reduces to model-finding. We present the language, related algorithms and complexity results as well as a prototypical implementation via answer set programming.

1 Introduction

In this work we tackle the problem of technical product configuration: Connect individual components conforming to a component catalogue so as to meet a given objective while respecting certain constraints. Solving such configuration problems is one of the major success stories of applied AI research: Starting from early work on rule based systems [16], 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 — for a survey see e.g. [14].

In the classical definition of a configuration problem the number of components to be used is fixed [19]. In practice, however, the number of components needed is often not easily stated beforehand. Hence extensions of the standard CSP formalism were developed, such as dynamic, composite or generative CSP [18, 21, 23], allowing to model the dynamic activation of components during search. Dynamic and composite CSP reduce to classical CSP [24] as the number of additional components is explicitly bounded; generative CSP allow infinite configurations.

Complementary to the CSP formalism and its variations there has also been substantial research on logic-based configuration 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. The first such logic-based formalisms were based on DLs [4, 17], reducing the problem of finding a configuration to constructing a model of a set of logical axioms. For DLs, the so-called tree model property prohibits modelling configurations where component connections form non-tree structures and, in general, the models need not be finite. The logic-based version of generative CSP presented in [9] also admits infinite models. In [12] it has been proposed to model the conditional inclusion of components by evaluating a positive existentially quantified first order sentence formed using conjunction and a restricted form of implication over an extensional finite constraint database. This work is closely related to dynamic and composite CSP, and hence does not come with support for describing component ports and connections. In [1] LoCo has been introduced. LoCo is also a fragment of first order logic, allows to describe arbitrary component topologies and implicitly bounds the number of components needed through the axioms and a number of explicitly bounded components. Configurations are found via model construction. The standard use case of LoCo looks as follows:

- The user specifies the problem in LoCo; cf. Section 3[4].
- It is then decided whether the specified problem is finite, and, if not, possible fixes are suggested. Finally bounds on the number of components are computed (3); cf. Section 4.
- Then the specification is translated to executable code. In Section 5 we touch upon a translation into answer set programming.

The present work extends [1] as follows: We extend the language of LoCo by axiom types (6) and (7). Next we present a stronger version of Proposition 1 as well as complexity results related to enforcing finite configurations. We show how integer programming can be used for bounds computation instead of the propagation algorithm presented in [1]. Finally we present a prototypical implementation.

2 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 [3]. This is a disguised rack configuration problem, a layered version of bin packing with side constraints.

The task is to put things of various types and sizes into cabinets which have to be stored in rooms of the house. 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. The goal is to find a minimal number of cabinets, counting twice all big cabinets.

3 The Language of 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 $\mathit{Component}(\mathit{id}, \vec{x})$. Here id is the component's identifier, and $\vec{x}$ a vector of further component attributes.

[1] Work funded by EPSRC Grant EP/G055114/1 Constraint Satisfaction for Configuration: Logical Fundamentals, Algorithms and Complexity

[2] University of Oxford, UK, email: firstname.lastname@cs.ox.ac.uk

[3] University of Hannover, Germany, email: vollmer@thi.uni-hannover.de

[4] Eventually this shall be done via a graphical user interface.

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. [7]). 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 3.1.

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

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

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.

Connections: Configuration is about connecting components: For every set $\{C_1, C_2\}$ of potentially connected components we introduce one of the binary predicate symbols $C_1 2 C_2$ and $C_2 2 C_1$, where predicate $C_i 2 C_j$ is of sort $\mathrm{ID}_i \times \mathrm{ID}_j$. We allow connections from a component type to itself, i.e., $C2C$. Connections between two component types are axiomatized as follows:[5]

$$(\forall id_1, \vec{x})\ C_1(id_1, \vec{x}) \Rightarrow \qquad (1)$$
$$(\exists_{l_1}^{u_1} id_2)\ [C_1 2 C_2(id_1, id_2) \wedge C_2(id_2, \vec{y}) \wedge \phi(id_1, id_2, \vec{x}, \vec{y})]$$

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 $\sum n \leq \mathit{Capacity}$. For these constraints we allow ϕ to be a Boolean combination of arithmetic expressions and attribute comparisons ($<, =, \ldots$).

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

Example 1. *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:*

$$(\forall id_{TA}, tSize, tBig)\ thingA(id_{TA}, tSize, tBig) \Rightarrow$$
$$(\exists_1^1 id_C)\ thingA2Cab(id_{TA}, id_C) \wedge cab(id_C, cSize, cBig, cTop) \wedge$$
$$[(cBig = tBig) \vee tBig = 0]$$

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

$$(\forall id_1, \vec{x})\ C_1(id_1, \vec{x}) \Rightarrow \qquad (2)$$
$$[\bigvee_i (\exists_{l_i}^{u_i} id_2)\ [C_1 2 C_2(id_1, id_2) \wedge C_2(id_2, \vec{y}) \wedge \phi_i(id_1, id_2, \vec{x}, \vec{y})]],$$

where the intervals $[l_i, u_i]$ are non-overlapping and $\phi_i(id_1, id_2, \vec{x}, \vec{y})$ may be a different formula for each case.[6] An even higher level of granularity can be reached by completely unfolding the existential counting quantifiers.

Example 2. *When connecting positions and cabinets we wish to differentiate between the cases where exactly one or two cabinets are connected to a position:*

$$(\forall id_P)\ pos(id_P) \Rightarrow$$
$$[(\exists_1^1 id_C)\ cab2Pos(id_C, id_P) \wedge cab(id_C, size, big, top)$$
$$\wedge\ top = 0] \vee$$
$$[(\exists_2^2 id_C)\ cab2Pos(id_C, id_P) \wedge cab(id_C, size, big, top) \wedge$$
$$big[1] = 0 \wedge big[2] = 0 \wedge$$
$$[(top[1] = 1 \wedge top[2] = 0) \vee (top[1] = 0 \wedge top[2] = 1)]]$$

Each case has a separate constraint part ϕ. When $l = u$ in the counting quantifier we can address each component instance and the respective attributes individually, abbreviated here as e.g. $big[1]$ and $big[2]$. While the order of the instances is not defined there clearly exists a permutation such that the constraint is satisfied.

Whenever possible an axiom for the reverse direction should be included, too:

$$(\forall id_2, \vec{x})\ C_2(id_2, \vec{x}) \Rightarrow \qquad (3)$$
$$(\exists_{l_2}^{u_2} id_1)\ [C_1 2 C_2(id_1, id_2) \wedge C_1(id_1, \vec{y}) \wedge \psi(id_1, id_2, \vec{x}, \vec{y})]$$

We stipulate that the upper bound u 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.

Next there are also rules for supporting one-to-many connections (4), i.e. connecting one component with a set of components.

$$(\forall id_1, \vec{x})\ C(id_1, \vec{x}) \Rightarrow \qquad (4)$$
$$(\exists_l^u id_2)\ [\bigvee_i C2C_i(id_1, id_2) \wedge C_i(id_2, \vec{y})]$$

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.

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

Example 3. *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:*

$$
\begin{aligned}
&(\forall id_C, cSize, cBig, top)\ cab(id_C, cSize, cBig, top) \Rightarrow \\
&\quad [(\exists_1 id_T)\ [thingA2Cab(id_T, id_C) \land thingA(id_T, tSize, tBig)] \lor \\
&\quad [thingB2Cab(id_T, id_C) \land thingB(id_T, tSize, tBig)]
\end{aligned}
$$

The exclusive-or variant of the axiom looks as follows, with l, u the same in all disjuncts:

$$(\forall id, \vec{x})\ C(id, \vec{x}) \Rightarrow \bigoplus_i [(\exists_l^u id_i) C2C_i(id, id_i) \land C_i(id_i, \vec{y})] \quad (5)$$

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.

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 form of a connection axiom:

$$
\begin{aligned}
&(\forall id, \vec{x})\ C(id, \vec{x}) \Rightarrow \qquad (6)\\
&[\bigvee_i [\bigwedge_j (\exists_{n_{i_j}}^{n_{i_j}} id_j) C2C_j(id_1, id_j) \land C_j(id_j, \vec{y}_j)] \land \phi_i(id, id_j, \vec{x}, \vec{y}_j)]
\end{aligned}
$$

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. 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.

As a last type of connection axiom we introduce a "connection-generating" axiom for expressing that some connections depend on the presence of others:

$$(\forall)\ \phi(\vec{x}) \Rightarrow C_1 2C_2(id_1, id_2). \quad (7)$$

Here $\phi(\vec{x})$ is a Boolean combination of components, connections and arithmetic and attribute comparisons.

Example 4. *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.*

$$
\begin{aligned}
&(\forall)\ [pers(id_{PE}) \land thingA(id_{TA}, \vec{attr}_T) \land pers2Thing(id_{PE}, id_{TA}) \land \\
&\quad cab(id_C, \vec{attr}_C) \land thingA2Cab(id_{TA}, id_T) \land pos(id_{PO}) \land \\
&\quad cab2Pos(id_C, id_{PO}) \land room(id_R) \land pos2Room(id_{PO}, id_R)] \Rightarrow \\
&\qquad room2Pers(id_R, id_{PE})
\end{aligned}
$$

3.1 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.

3.1.1 Domain Knowledge

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.
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.

Example 5. *Position is used as a component port of a room to place cabinets in it at a certain location. The connection of a component port has the same structure as a binary connection axiom:*

$$
\begin{aligned}
&(\forall id_R)\ room(id_R) \Rightarrow \\
&\quad (\exists_1^4 id_P)\ room2Pos(id_R, id_P) \land pos(id_P)
\end{aligned}
$$

Attribute Ranges For all attribute sorts a naming-scheme is included. For ordinary component attributes these take the form (8) for sort predicate T and some first order formula $\phi(x)$:

$$(\forall x)\ T(x) \equiv \phi(x). \quad (8)$$

For component attributes of sort ID the naming-scheme has the form (9); i.e. components are numbered:

$$(\forall x) T(x) \Rightarrow (\exists n) x = \text{TName}(n). \quad (9)$$

The form (9) 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

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

expressing the fact that, in database terminology, the respective ID is a *key*. Unique name axioms for all distinct 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 For each component type the catalogue contains information on the instances that actually can be manufactured. In LoCo this is done with an axiom:

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

where the $\vec{V}_i$ are vectors of ground terms. If the component has no attributes the axiom is omitted.

3.1.2 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:

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

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 ground literals or via formulas of the form $(\exists)C(id, \vec{x})$ or $(\forall)\neg C(id, \vec{x})$, where $id, \vec{x}$ may be variables or terms. Known (non-)connections can be specified via ground literals like e.g. $\neg C_1 2C_2(\mathrm{ID}_1, \mathrm{ID}_2)$. Similar to input components we support closure axioms on connections

$$(\forall)\ C_i 2C_j(id_i, id_j) \equiv \bigvee (id_i = \mathrm{ID}_1 \wedge id_j = \mathrm{ID}_2).$$

4 Enforcing Finite Configurations

Next we discuss how to enforce that configurations contain only finitely many components.

4.1 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. Let us first introduce some notation: Let $\mathcal{C}$ denote the set of components of type C that can be used in a configuration. Let $|\mathcal{C}|$ denote this set's cardinality and $\mathrm{lb}(\mathcal{C})$ and $\mathrm{ub}(\mathcal{C})$ the lower and upper bound on the set's cardinality. Then assume a binary connection defined by formulas (1) and (3). For component C_2 we then have:

$$l_1 * \mathrm{lb}(\mathcal{C}_1) \leq u_2 * |\mathcal{C}_2| \text{ and } l_2 * |\mathcal{C}_2| \leq u_1 * \mathrm{ub}(\mathcal{C}_1) \qquad (10)$$

Connecting the elements of $\mathcal{C}_2$ to as many elements of $\mathcal{C}_1$ as possible (u_2) while making only the minimum number of connections in the backwards direction (l_1) gives a lower bound on the cardinality of $\mathcal{C}_2$ if we assume $|\mathcal{C}_1|$ to be as small as possible. The intuition behind the upper bound is analogous. Observe, however, that we cannot derive the desired finite bound if $l_2 = 0$ or $\mathcal{C}_1$ is not finitely bounded. We disregard the "constraint formulas" ϕ and ψ for this calculation.

Next assume we have a basic one-to-many connection axiom (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. Here we get:

$$\sum_i l_i * \mathrm{lb}(\mathcal{C}_i) \leq u * |\mathcal{C}| \text{ and } l * |\mathcal{C}| \leq \sum_i u_i * \mathrm{ub}(\mathcal{C}_i) \qquad (11)$$

In the case of an exclusive disjunction in the one-to-many axiom (5) we get:

$$\sum_i \mathrm{lb}(x_i) \leq |\mathcal{C}| \leq \sum_i \mathrm{ub}(x_i) \text{ with} \qquad (12)$$
$$l_i * \mathrm{lb}(\mathcal{C}_i) \leq x_i * u \text{ and } x_i * l \leq u_i * \mathrm{ub}(\mathcal{C}_i)$$

The number of times case i applies is reflected by x_i. We observe that for both formulas (11) and (12) we need $l > 0$ and all the $\mathcal{C}_i$ to be finitely bounded for $\mathcal{C}$ to be finitely bounded, too.

Next consider a general one-to-many axiom (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. Denote by x_i the number of times case i applies. Then we have for

$$\sum_i \mathrm{lb}(x_i) \leq |\mathcal{C}| \leq \sum_i \mathrm{ub}(x_i) \text{ with} \qquad (13)$$
$$l_j * \mathrm{lb}(\mathcal{C}_j) \leq \sum_i x_i * n_{i_j} \leq u_j * \mathrm{ub}(\mathcal{C}_j)$$

Equations similar to the ones we just presented can also be found in [8], a work that proposes (1) to model configuration problems via UML and (2) to solve them via integer programming. We note that LoCo is considerably more general, though.

4.2 Globally Bounding Component Numbers

We formalize these local interactions between different component types via the configuration graph. This 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 local condition (10) with $l_2 > 0$ we include an edge from C_2 to C_1. For local conditions (11) and (12) we include an and-edge from C to all C_i if $l > 0$. If we have an axiom (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* [6], 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.

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 [1]: 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 [13].

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:. Finding a minimal key for a database under functional dependencies is NP-complete [15]. 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 $C \rightarrow B$ becomes an implication $(\bigwedge \mathcal{C}) \Rightarrow (\bigwedge \mathcal{B})$; i.e. it can be expressed as Horn clauses. □

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.

4.3 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 (10), (11), (12) and (13) 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, an arbitrary integer programming problem can be reduced to a LoCo problem giving rise to condition (13) and we have:

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

But just how many components can we have in the worst case? Assume we have $2n$ binary connection axioms forming a path $(C_1, C_2, \ldots, C_n)$ in the configuration graph, with C_n the only input component and u the same in all axioms. We can then generate up to u^n instances of C_1, i.e. exponentially many. In this context it is worth pointing out that cycles in the graph can lead to smaller, but not to larger, upper bounds.

While this exponential blow-up can not be avoided we could ease the task of bounds computation by settling for less tight bounds: For each generated component we can read off an upper bound from the longest path in the configuration path leading to an input component. But tighter bounds of course mean a smaller search space after translating LoCo problems into executable formats such as constraint programming or SAT.

5 Implementing LoCo

The major objective 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.

We have prototypically implemented LoCo in answer set programming using the Potassco framework [11]. A detailed description of the translation is beyond the scope of this paper and, together with a thorough evaluation, subject of future work.

Example 6. *The following code snippet shows the transformation of a binary connection, one of the cornerstones of our formalism:*

```
1{thingA2Cab(T,C):cabGen(C)}1 :- thingA(T).
1{thingA2Cab(T,C):thingA(T)}cMax :- cab(C).
:- thingA2Cab(T,C), not cab(C).
:- not c1(T,C), thingA2Cab(T,C).
c1(T,C) :- thingA2Cab(T,C), tBig(T,B), B==0.
c1(T,C) :- thingA2Cab(T,C),
           tBig(T,TB), cBig(T,CB), TB==CB.
```

Line 1 represents the connection from thing to cabinet as shown in Example 1 while line 2 represents the reverse direction. Both lines use so-called cardinality constraints [22]: Line 1 means that there is exactly one ground instance of the predicate `thingA2Cab(T,C)` *for every* `T` *such that* `C` *and* `T` *are identifiers of cabinets and things of type A. The condition part (*`cabGen(C)`*) in such rules must be specified by ground facts in the knowledge base. Hence our knowledge base contains all instances of* `thingA` *and* `cabGen`*; the latter are the finitely many component instances which* might *be used in the configuration. The instances of* `cab` *are those that actually* are *used. The integrity constraint in line 3 ensures that every cabinet that features in a connection also is in the extension of the* `cab` *predicate. Lines 4-8 depict the mapping of the constraint part (see Example 1). The integrity constraint in line 4 states that for every connection between a thing and a cabinet the constraint* `c1` *must hold. The following lines represent the mapping of a disjunction, i.e. either thing is not big (line 5) or thing and cabinet have the same value for attribute big (lines 6-7). Using similar transformation steps we furthermore are able to map arbitrary Boolean combinations of arithmetic expressions, attribute comparisons and aggregate functions.*

The translation of the other axiom types of LoCo is along the same lines, but considerably more involved. Note that we explicitly represent all generated components that might be used in the configuration (there is a finite upper bound); however, this may be exponentially many in the size of the domain axiomatization.

After translating LoCo problems to answer set programs as sketched above deciding satisfiability of the ground programs is known to be NP-complete [22]. For non-ground programs, however, this bound does not hold; in fact, answer set programming with variables is NEXPTIME-complete [5].

Preliminary experimental results are very encouraging, however: For the House Problem we can compete with the hand-written problem encoding in answer set programming presented in [10]; our translation yields a very similar program. On the Partner Units Problem, another challenging configuration problem, we reach the same performance as the answer set program presented in [2] if for the latter the problem-specific search strategy is turned off.

6 Future Work

The big open theoretical question of this work is whether the task of deciding satisfiability of LoCo problems is in NP. A positive answer to this question would pave the way to an implementation that avoids generating exponentially many components.

For practical usability developing an intuitive graphical user interface will be crucial. Finding optimal configurations is likewise of great practical importance. Here we envision a model that attaches costs to individual component instantiations and the respective connections and allows to build objective functions on top of that.

Finally we plan to translate LoCo into the MINIZINC language next [20]. This will give us access to state-of-the-art SAT, constraint, integer programming and hybrid solvers and provide LoCo with a portfolio of complementary solving back-ends.

REFERENCES

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

[2] M. Aschinger, C. Drescher, G. Gottlob, G. Friedrich, P. Jeavons, A. Ryabokon, and E. Thorstensen, 'Optimization Methods for the Partner Units Problem', in *Proceedings of the 8th International Conference on the Integration of Artificial Intelligence and Operations Research Techniques into Constraint Programming for Combinatorial Optimization Problems (CPAIOR), Berlin, Germany*, (2011).

[3] M. Bettex, A. Falkner, W. Mayer, and M. Stumptner, 'On Solving Complex Rack Configuration Problems using CSP Methods', in *Configuration Workshop at the 21st International Conference on Artificial Intelligence (IJCAI), Pasadena, California*, (2009).

[4] M. Buchheit, R. Klein, and W. Nutt, 'Constructive Problem Solving: A Model Construction Approach towards Configuration', Technical Report TM-95-01, DFKI, (1995).

[5] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov, 'Complexity and Expressive Power of Logic Programming', *ACM Computing Surveys*, **33**(3), 374–425, (September 2001).

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

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

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

[9] G. Friedrich and M. Stumptner, 'Consistency-Based Configuration', in *Configuration Workshop at the 16th National Conference on Artificial Intelligence (AAAI), Orlando, Florida*, (1999).

[10] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(Re)configuration Based on Model Generation', in *Proceedings of the Second Workshop on Logics for Component Configuration, (LoCoCo), Perugia, Italy*, (2011).

[11] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider, 'Potassco: The Potsdam Answer Set Solving Collection', *AI Communications*, **24**(2), 105–124, (2011).

[12] G. Gottlob, G. Greco, and T. Mancini, 'Conditional Constraint Satisfaction: Logical Foundations and Complexity', in *Proceedings of the 20th International Conference on Artificial Intelligence (IJCAI), Hyderabad, India*, (2007).

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

[14] U. Junker, 'Configuration', in *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, 837 – 874, Elsevier, (2006).

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

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

[17] D. L. McGuinness and J. R. Wright, 'Conceptual Modelling for Configuration: A Description Logic-based Approach', *AI EDAM*, **12**(4), 333–344, (1998).

[18] S. Mittal and B. Falkenhainer, 'Dynamic Constraint Satisfaction Problems', in *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI), Boston, Massachusetts*, (1990).

[19] S. Mittal and F. Frayman, 'Towards a Generic Model of Configuration Tasks', in *Proceedings of the 11th International Conference on Artificial Intelligence (IJCAI), Detroit, Michigan*, (1989).

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

[21] D. Sabin and E. C. Freuder, 'Configuration as Composite Constraint Satisfaction', in *Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop (AIMRP), Albuquerque, New Mexico*, (1996).

[22] Patrik Simons, Ilkka Niemelä, and Timo Soininen, 'Extending and Implementing the Stable Model Semantics', *Artificial Intelligence*, **138**(1–2), 181–234, (2002).

[23] M. Stumptner, A. Haselböck, and G. Friedrich, 'Generative Constraint-based Configuration of Large Technical Systems', *AI EDAM*, **12**(4), 307–320, (1998).

[24] E. Thorstensen, 'Capturing Configuration', in *Doctoral Program at the 16th International Conference on Principles and Practice of Constraint Programming (CP), St. Andrews, Scotland*, (2010).