LoCo — A Logic for Configuration Problems

MARKUS ASCHINGER, University of Oxford
CONRAD DRESCHER, University of Oxford
GEORG GOTTLOB, University of Oxford
HERIBERT VOLLMER, Leibniz University Hannover

In this work we present LoCo, a fragment of classical first order logic carefully tailored for expressing technical product configuration problems. The core feature of LoCo is that the number of components used in configurations does not have to be finitely bounded explicitly, but instead is bounded implicitly through the axioms. Computing configurations is equivalent to the task of model-finding. We present the language, related algorithms and complexity results as well as a prototypical implementation via answer set programming.

Categories and Subject Descriptors: I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods
General Terms: Languages, Algorithms, Theory
Additional Key Words and Phrases: Technical product configuration, model-based reasoning

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. TECHNICAL PRODUCT CONFIGURATION AND APPLIED ARTIFICIAL INTELLIGENCE

This work deals with the problem of technical product configuration. Solving such configuration problems is one of the major success stories of applied AI research: The early work on using rule based configurators for customizing computers [McDermott 1982] is generally seen as the field’s starting point. Since then manifold general purpose AI techniques such as constraint satisfaction problem (CSP) and Boolean satisfiability (SAT) solving, heuristic search, and description logics (DLs) have successfully been applied to configuration. Practical applications are nowadays ubiquitous and range from configurable cars [Sinz et al. 2003] to the configuration of telephone communication switching units [Fleischanderl et al. 1998] — for surveys see e.g. [Sabin and Weigel 1998] or the more recent [Junker 2006]. By now it has become apparent that there are also manifold connections to the domain of software configuration, see e.g. [Hubaux et al. 2012].

1.1. The Problem of Technical Product Configuration and How to Solve It

The reference definition of a configuration problem has been given in [Mittal and Frayman 1989]:

Definition 1.1 (Configuration Problem). Given: A fixed, predefined set of components, where a component is described by a set of properties, ports for connecting it to other components, constraints at each port that describe the components that can be connected at that port, and other structural constraints, some description of the desired configuration and some criteria for making optimal selections.

Build: One or more configurations that satisfy all the requirements, where a configuration is a set
of components and a description of the connections between the components in the set, or, detect inconsistencies in the requirements.

Please note that this classical definition of a configuration problem leaves considerable wiggle room as for how exactly the problem is to be formalized. It is also worth pointing out that in this definition the number of components available to build the final configuration is fixed; that is, it cannot be changed during runtime.

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

1.1.2. Problem Formalizations and Reasoning Methods. Let us next highlight the diversity of the problem formalizations and reasoning methods that have successfully been applied to configuration. In its simplest form the problem is formalized as a standard CSP and existing mature solver technology is exploited. Contemporary research in this direction includes e.g. the compilation of the CSP for fast interactive reasoning [Amilhastre et al. 2002; Andersen et al. 2010] or the use of global constraints for greater deductive power [Karatas et al. 2010]. These approaches do not come with explicit support for ports or connections between components; also, the number of individual components available to build the final configuration has to be fixed prior to solving.

From a different point of view one may think about configuration as a dynamic process: The problem is solved by incrementally adding components and selecting values for their attributes until the overall requirements are met. In such an approach it is easier to express that the exact number of components to be included in the final configuration is not fixed but depends on the choices made for the previous components. This approach to configuration initially led to the development of extensions of the classic, static CSP formalism tailored to configuration such as dynamic, composite or generative CSP [Mittal and Falkenhainer 1990; Sabin and Freuder 1996; Stumptner et al. 1998], all of which allow us to model the dynamic activation of components during search. We observe that, because the number of components available is fixed, both dynamic and composite CSP polynomially reduce to classical CSP [Thorsten sen 2010] and hence are in some sense equivalent to them. Nevertheless, the associated dynamic reasoning methods seem to be promising in practice. Generative CSP, on the other hand, do not bound the number of components; here, however, it is not clear how to rule out infinite configurations. Of these formalisms, only generative CSP come with explicit support for ports and component connections.

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 [Buchheit et al. 1995; McGuinness and Wright 1998], reducing the problem of finding a configuration to constructing a model of a set of logical axioms. On the one hand, these approaches support the representation of component connections. On the other hand, the so-called tree model property of Description Logics is at odds with modelling configurations where the connections form non-tree structures. Also, in general, the models of a DL axiomatization need not be finite, whereas the technical products to be configured always are. Similarly, the logic-based
version of generative CSP presented in [Friedrich and Stumptner 1999] also admits infinite models / configurations. In [Gottlob et al. 2007] it has been proposed to model the conditional inclusion of components by evaluating a strictly positive, existentially quantified first order sentence formed by using conjunction and a restricted form of implication over an extensional finite constraint database. This work turns out to be a notational variant of dynamic and composite (and hence also standard) CSP [Thorsten 2010]; like these it does not feature support for ports or connections either. On the other hand, this work has pioneered the identification of families of such conditional configuration (optimization) problems that admit tractable reasoning/efficient processing.

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

In the most elaborate configuration formalisms, a form of iterative component refinement according to some taxonomy is interleaved with a form of conditional CSP solving [Mailharro 1998; Junker 2006]; reasoning about ports and component connections is also supported. Due to their intricate nature, however, for these formalisms problems such as termination of configuration finding appear not to be well understood theoretically yet.

1.2. The Case for LoCo

In this work we present LoCo, a logic that has carefully been tailored to meet the demands of technical product configuration. We identify configurations with models of the logic; hence, configuration finding is model finding. LoCo supports the notions of component ports and connections and allows us to describe arbitrary component topologies; it comes with a rich language for describing binary and one-to-many connections as well as constraints that must hold for connected components. Most importantly, it relaxes the requirement of placing explicit bounds on the number of components. Instead it implicitly bounds the number of components needed through the axioms and a given set of explicitly bounded components whenever possible. We employ existential counting quantifiers to indicate the number of possible connections from one component type to another; from these we derive the finite bounds. If finite bounds could not be inferred, we can derive a smallest fix for the problem: a set of components that — if bounded by hand — suffices to make the problem finite. As configurations are then guaranteed to be finite there are no fundamental obstacles to either fully automated or interactive reasoning support.

LoCo follows the same idea. In contrast to LoCo, however, these UML approaches only derive lower bounds on the number of available components and hence do not rule out arbitrarily large configurations. Such an upper bound is vital, however, if we want to defer reasoning e.g. to SAT or constraint solvers instead of the integer linear solvers used in [Falkner et al. 2010; Feinerer 2013].1 Another weakness of these approaches is their very limited support for component attributes and consequently for expressing constraints on these.

The standard use case of LoCo hence looks as follows:

— The user specifies the problem in LoCo; cf. Section 3.
— It is then decided whether the specified problem is finite (admits only finite models), and, if not, possible fixes are suggested.
— After that bounds on the number of components are computed; cf. Section 4.
— Finally, the specification is translated to executable code. LoCo allows translations into different target languages. In Section 6 we touch upon a translation into answer set programming.

1 Considering configurations/databases of arbitrary size has further theoretical consequences: For example, Lenzerini and Nobili can reduce their notion of strong satisfiability (a legal database instance with at least some non-empty relations exists) to the existence of a fully populated database (no empty relations allowed). In LoCo’s approach this would not work as we cannot add arbitrarily many components.
The reasoning problems that we study are hence the following: (1) Decide whether the problem is finite. (2) Compute a smallest set of components sufficient to make the problem finite (if necessary). (3) Find a model/configuration. As LoCo allows the user to specify partial configurations to be used as starting point the task of interactive configuration can be reduced to a series of configuration finding problems. The feature of specifying partial configurations as part of the input can also be used to support a limited form of reconfiguration; this works as long as there is no conflict/inconsistency between the (partial) legacy instance and the new constraint/objective. Dealing with the case of an actual inconsistency requires further research and is a promising subject of future work as is the reasoning task of explaining configuration results to end-users.

LoCo has originally been introduced in [Aschinger et al. 2011], and in [Aschinger et al. 2012] we have elaborated upon the above mentioned reasoning tasks. In this paper we summarize the contributions of these two references and add the following new contributions: First we provide an answer to the following fundamental question that we previously had to leave open: What is the computational complexity of deciding whether a LoCo axiomatization is satisfiable, or, equivalently, how hard is it to decide whether there is a configuration satisfying the requirements (Proposition 5.1)? We also provide a further new result (Proposition 4.5) concerning the tightness of the bounds on the number of components to be used in configurations.

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 [Bettex et al. 2009]. The House Problem reflects constraints and properties that can be found in a wide range of problems involving the design and assembly of complex systems and software processes. It is basically a toy problem derived from a real-world configuration problem in close analogy to the rack configuration problem; a layered version of bin packing with side constraints.

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

The task of the House Problem is to put things of various types and sizes into cabinets which have to be stored in rooms of the house. For brevity and the sake of illustration, we cover only certain parts of the problem that we think are particularly helpful in understanding the underlying formalism. A cabinet has two shelves, each providing a certain storage space for either things of type A or B. Constraints on component attributes determine where a thing or a cabinet can be stored: Big things can only be stored in big cabinets whereas some cabinet need to be located at a certain position in a room; in the case of two small cabinets one can possibly be placed on top of the other in the same position. Every thing is owned by a person and things of different persons cannot be placed together in the same room. The goal is to find a minimal number of cabinets, counting twice all big cabinets.

Figure 1 depicts an example scenario for the house problem, including the basic (binary) connections between components together with cardinalities restricting the number of potential connections. Connections marked as \textit{(Input)} are pre-defined, i.e. they are already specified in a given problem instance and therefore part of the input.

3. THE LANGUAGE OF LOCO
In this section we first introduce the formal language of LoCo and then discuss its use for specifying configuration problems.
3.1. Introducing LoCo as Fragment of FO

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 $\text{Component}(id, \vec{x})$. Here $id$ is the component’s identifier, and $\vec{x}$ a vector of further component attributes.

**Sorted Attributes:** The component attributes belong to different sorts — e.g. numbers, strings, etc.. Using sorted variables and terms simplifies notation. In particular, for each component type we introduce one sort $I_D$ 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 similar to the reduction of classical many-sorted logic to pure first order logic (cf. e.g. [Enderton 1972]). We first introduce unary predicates for each sort (e.g. $ID$ for sort $I_D$) and add domain partitioning axioms:

$$(\forall x) \bigvee_{S \in \text{SORTS}} S(x),$$

$$(\forall x) \bigwedge_{S_i, S_j \in \text{SORTS}, i \neq j} \neg(S_i(x) \land S_j(x)).$$

Then, in a sorted formula, we replace each subformula $(\forall id) \phi(id)$, where the universal quantifier ranges over component identifiers only, by $(\forall x) \text{ID}(x) \Rightarrow \phi(x)$ and likewise $(\exists id) \phi(id)$ by $(\exists x) \text{ID}(x) \land \phi(x)$ — this is the standard reduction from many-sorted to classical FO. We postpone the discussion of how to treat sorted terms until Section 3.2.

**Counting Quantifiers:** For restricting the number of potential connections between components we use existential counting quantifiers $\exists^{|} x$ with lower and upper bounds $l$ and $u$ such that $l \leq u$, $l \geq 0$ and $u > 0$. For example, a formula $(\exists^{|} u x) \phi(x)$ 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
As usual sorted quantifiers range over a single sort only. But occasionally, by an abuse of notation, we will write e.g. $(\exists x) \phi(x) \lor \psi(x)$, where $\phi$ and $\psi$ expect different sorts. This abbreviates a formula enforcing that the total number of objects satisfying formulas $\phi$ or $\psi$ is between $l$ and $u$, where the disjunction is inclusive.

**Connection Axioms:** Configuration is about connecting components: For every set \( \{C_1, C_2\} \) of potentially connected components we introduce one of the binary predicate symbols \( C_12C_2 \) and \( C_22C_1 \), where predicate \( C_i2C_j \) is of sort \( \text{Id}_i \times \text{Id}_j \). \(^2\) We allow connections from a component type to itself, i.e., \( C2C \). Connections between two component types are axiomatized as follows:\(^3\)

\[
(\forall id_1, \vec{x}) \ C_1(id_1, \vec{x}) \Rightarrow \\
(\exists_i^{\mathbb{N}} \ id_2) \big[ C_12C_2(id_1, id_2) \land C_2(id_2, \vec{y}) \land \phi(id_1, id_2, \vec{x}, \vec{y}) \big]
\]

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 \( \phi \) (with variables among \( id_1, id_2, \vec{x}, \vec{y} \)) is to express additional constraints, like e.g. an aggregate function \( \sum n \leq \text{Capacity} \). For these constraints we allow \( \phi \) to be a Boolean combination of arithmetic expressions and attribute comparisons \((<, =, \ldots)\) over a subset of all quantified variables of the axiom. Generally the type of supported subformulas is restricted to the expressiveness of the chosen target output language. For special constructs like e.g. aggregate functions \( \text{SUM} \) and \( \text{COUNT} \) we perform language-specific transformations. For these constraints we allow \( \phi \) to be a Boolean combination of arithmetic expressions and attribute comparisons \((<, =, \ldots)\) over a subset of all quantified variables of the axiom. Generally the type of supported subformulas is restricted to the expressiveness of the chosen target output language. For special constructs like e.g. aggregate functions \( \text{SUM} \) and \( \text{COUNT} \) we perform language-specific transformations.

**Example 3.1.** 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 — in configuration terms big things and small cabinets are not compatible:

\[
(\forall id_{TA}, t\text{Size}, t\text{Big}) \ \text{thing}A(id_{TA}, t\text{Size}, t\text{Big}) \Rightarrow \\
(\exists_i^{\mathbb{N}} \ id_C) \big[ \text{thing}A2C(id_{TA}, id_C) \land \text{c}ab(id_C, c\text{Size}, c\text{Big}, c\text{Top}) \land \\
[ (c\text{Big} = t\text{Big}) \lor (t\text{Big} = 0) ] \big]
\]

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

\[
(\forall id_1, \vec{x}) \ C_1(id_1, \vec{x}) \Rightarrow \\
\big[ \bigvee_i (\exists_i^{\mathbb{N}} \ id_2) \big[ C_12C_2(id_1, id_2) \land C_2(id_2, \vec{y}) \land \phi_i(id_1, id_2, \vec{x}, \vec{y}) \big] \big],
\]

\(^2\)Note that this precludes having multiple different connection relationships between two different component types.

\(^3\)Throughout this paper free variables in formulas are to be read as existentially quantified.
where the intervals \([l_i, u_i]\) are non-overlapping in order to be mutually exclusive and \(\phi_i(id_1, id_2, \vec{x}, \vec{y})\) may be a different formula for each case.\(^4\) An even higher level of granularity can be reached by completely unfolding the existential counting quantifiers, i.e. defining a separate case for each possible number of occurring \(id_2\) objects.

**Example 3.2.** When connecting positions and cabinets we wish to differentiate between the cases where exactly one or two cabinets are connected to a position:

\[
(\forall id_P) \text{pos}(id_P) \Rightarrow \\
\left( (\exists_{id_{C}}) \text{cab2Pos}(id_{C}, id_P) \land \text{cab}(id_{C}, \text{size}, \text{big}, \text{top}) \land \right.
\left. \left( \text{top} = 0 \right) \right) \lor \\
\left( (\exists_{id_{C}}) \text{cab2Pos}(id_{C}, id_P) \land \text{cab}(id_{C}, \text{size}, \text{big}, \text{top}) \land \right.
\left. \left( \text{big} = 0 \land ( (\text{top}[1] = 1 \land \text{top}[2] = 0) \lor (\text{top}[1] = 0 \land \text{top}[2] = 1) ) \right) \right)
\]

Each case has a separate constraint part \(\phi\). When \(l = u\) in the counting quantifier we can address each component instance and the respective attributes individually, abbreviated here as e.g. \(\text{big}[1]\) and \(\text{big}[2]\). 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:

\[
(\forall id_2, \vec{x}) \text{C}_2(id_2, \vec{x}) \Rightarrow \\
\left( (\exists_{id_{1}}) \left[ \bigwedge_{i} \left[ \text{C}_i(2)(id_1, id_2) \land \text{C}_i(id_2, \vec{y}) \right] \land \phi(id_1, id_2, \vec{x}, \vec{y}) \right] \right)
\]

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.

\[
(\forall id_1, \vec{x}) \text{C}(id_1, \vec{x}) \Rightarrow \\
\left( (\exists_{i} id_2) \left[ \bigvee_{i} \left[ \text{C}_i(2)(id_1, id_2) \land \text{C}_i(id_2, \vec{y}) \right] \land \phi(id_1, id_2, \vec{x}, \vec{y}) \right] \right)
\]

In this rule the quantifier \(\exists_{i} id_2\) 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. This condition is necessary to guarantee the correct computation of component bounds and will be described in more detail in section 4.1.

**Example 3.3.** 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:

\(^4\) Note that there are unique smallest, and biggest, \(l_i\), and \(u_i\), respectively.
The next example highlights how to combine binary and one-to-many connection axioms in order to model the common configuration task of resource balancing.

Example 3.4. Assume that things of type A contribute a certain amount of some resource whereas things of type B consume this resource. The exact quantities will be described in the component catalogue (to be introduced below). We want to ensure that for each cabinet the amount of the resource contributed is greater or equal to that consumed. To this end, for both cabinets and things of type A and B we introduce an additional numerical attribute — for better readability we are going to ignore the other component attributes. As before, the binary connection axioms describe how many things of type A and B can be stored per cabinet, say between one and two each. The following one-to-many axiom ensures the resource-balancing:

\[
(\forall \text{id}_C \text{ cab}(\text{id}_C, \text{cSize}, \text{cBig}, \text{top}) \Rightarrow
(\exists_1 \text{id}_T) \left[ \left[ \text{thingA2Cab}(\text{id}_T, \text{id}_C) \land \text{thingA}(\text{id}_T, \text{tSize}, \text{tBig}) \right] \lor \left[ \text{thingB2Cab}(\text{id}_T, \text{id}_C) \land \text{thingB}(\text{id}_T, \text{tSize}, \text{tBig}) \right] \right]
\]

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

\[
(\forall \text{id}, \vec{x}) \quad C(\text{id}, \vec{x}) \Rightarrow \bigoplus_i \left[ \left( \exists_i^{l} \text{id}_i \right) \left[ C^2C(\text{id}, \text{id}_i) \land C(\text{id}_i, \vec{y}_i) \right] \right]
\] (5)

This allows the natural formulation of certain compatibility relations that otherwise would have to be formulated in LoCo’s standard way for expressing compatibility relations: By using constraints attached to connection axioms.

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:

\[
(\forall \text{id}, \vec{x}) \quad C(\text{id}, \vec{x}) \Rightarrow \bigvee_i \left[ \left( \exists_i^{n_{ij}} \text{id}_j \right) \left[ C^2C^i(\text{id}, \text{id}_i) \land C(\text{id}_i, \vec{y}_i) \right] \right] \land \phi_i(\text{id}, \text{id}_j, \vec{x}, \vec{y}_j)
\] (6)

The component \(C\) can be connected to a number of components \(C_j\) — but \(C\) cannot be among the \(C_j\). The rule has \(i\) cases: Each case \(i\) states for each of the components \(C_j\) the exact number \(n_{ij}\) of connections between \(C\) and \(C_j\). Note that we allow \(n_{ij} = 0\), but there must not be two disjuncts with identical bounds \(n_{ij}\) 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:

\[(\forall \phi(\vec{x})) \Rightarrow C_1 2 C_2 (id_1, id_2)\]  

(7)

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

**Example 3.5.** 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.

\[
(\forall \left[ \text{pers}(id_{PE}) \land \text{thingA}(id_{TA}, \text{attr}_{TA}) \land \text{pers2Thing}(id_{PE}, id_{TA}) \land \\
\text{cab}(id_C, \text{attr}_{C}) \land \text{thingA2Cab}(id_{TA}, id_{T}) \land \text{pos}(id_{PO}) \land \\
\text{cab2Pos}(id_C, id_{PO}) \land \text{room}(id_R) \land \text{pos2Room}(id_{PO}, id_R) \right] \Rightarrow \\
\text{room2Pers}(id_R, id_{PE})
\]

### 3.2. 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.

**3.2.1. 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. Let us next briefly elaborate on how to model the concept of a port in LoCo.

**Ports** Component ports are modelled as individual components in LoCo. A normal component may have many ports (i.e. be connected to many port components); however, each port belongs to exactly one component.

**Example 3.6.** 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:

\[
(\forall id_R) \text{room}(id_R) \Rightarrow \\
(\exists id_P) \left[ \text{room2Pos}(id_R, id_P) \land \text{pos}(id_P) \right]
\]

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

\[
(\forall x) S(x) \equiv \phi(x).
\]  

(8)

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

\[(\forall x) S(x) \Rightarrow (\exists n) x = \text{SName}(n). \tag{9}\]

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

\[(\forall \text{id}, \vec{x}) C(\text{id}, \vec{x}) \equiv \bigvee_i \vec{x} = \vec{V}_i, \tag{10}\]

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

**Component Catalogue (v2)** The component catalogue as outlined above and introduced in [Aschinger et al. 2012] does not conform to industrial practice in that it is extensional: Every legal combination of attribute values per component type has to be explicitly listed. In practice, however, the component catalogue is usually specified via attribute ranges and constraints that determine the legal combinations [Junker 2006]. This kind of component catalogue can be expressed in LoCo as an axiom

\[(\forall \text{id}, \vec{x}) C(\text{id}, \vec{x}) \equiv \phi(\vec{x}), \tag{11}\]

where \(\phi(\vec{x})\) is a quantifier-free formula on attribute comparisons.

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

\[(\forall x) S(x) \equiv \bigvee_i x = V_i, \]

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

3.2.2. 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:

\[(\forall x) \text{ID}(x) \equiv \bigvee_{\text{id}_i \in \mathcal{ID}} x = \text{ID}_i.\]
where $\mathcal{ID}$ is a finite set of identifiers $\text{ID}_i$ and $\text{ID}$ 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 $\text{EXCESS}$.

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. $\neg C_12C_2(ID_1, ID_2)$. Similar to input components we support closure axioms on connections

$$(\forall) C_j2C_j(id_i, id_j) \equiv \bigvee (id_i = \text{ID}_1 \land id_j = \text{ID}_2).$$

We summarize the above discussion of all the different LoCo features in the following definition.

Definition 3.7 (Configuration Domain Axiomatization in LoCo). A configuration domain axiomatization in LoCo consists of domain knowledge and instance knowledge. The domain knowledge comprises

— connection axioms in the forms (1, 2, 4 — 7);
— a specification of the attribute ranges (8);
— a specification of the component identifier naming scheme (9);
— a component catalogue in either of the forms (10) or (11); and
— an axiomatization of finite domain arithmetic

whereas the instance knowledge is made up of

— a designation of the input components together with their respective count;
— partial configurations, consisting of components and connections with possibly existentially quantified attributes; and
— forbidden partial configurations, consisting of components and connections with possibly universally quantified attributes.

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. We disregard the “constraint formulas” $\phi$ and $\psi$ for this calculation; the bounds are based only on the lower and upper bounds on the number of connections expressed in the existential counting quantifiers.

Let us first introduce some notation: Let $C$ denote the set of components of type $C$ that can be used in a configuration and let $|C|$ denote this set’s cardinality. Then assume a binary connection defined by formulas (1) and (3). For component $C_2$ we then have:

$$l_1 * |C_1| \leq u_2 * |C_2| \text{ and } l_2 * |C_2| \leq u_1 * |C_1| \quad (12)$$

The first inequality holds because we cannot connect the elements of $C_2$ to more than $u_2$ elements of $C_1$ each, while each element of $C_1$ has to be connected to at least $l_1$ elements of $C_2$. Hence, if we have a lower bound on the cardinality of $C_1$ this implies a lower bound on the cardinality of $|C_2|$. The intuition behind the upper bound is analogous. Here in particular, if we have a finite upper bound on the cardinality of $C_1$ and $l_2 \neq 0$ then we can derive a finite upper bound on the cardinality of $C_2$.

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:
\[
\sum_i l_i \cdot |C_i| \leq u \cdot |C| \text{ and } l \cdot |C| \leq \sum_i u_i \cdot |C_i|, \tag{13}
\]

as each element of \( C_i \) has to be connected to at least \( l_i \) elements of \( C \), whereas each of the latter can be connected to at most \( u \) elements of \( \bigcup C_i \).

In the case of an exclusive disjunction in the one-to-many axiom (5) each element of \( C \) can be connected to elements of one of the \( C_i \) only. Let \( x_i \) denote the number of times some element of \( C \) uses \( C_i \) for its connections. Then we get for all \( i \):

\[
\sum_i x_i = |C| \text{ with } l_i \cdot |C_i| \leq x_i \cdot u \text{ and } x_i \cdot l \leq u_i \cdot |C_i|, \tag{14}
\]

We observe that for both formulas (13) and (14) we need both \( l > 0 \) and all the \( C_i \) to be finitely bounded in order to derive a finite bound on \( C \).

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 \( C \). Again, denote by \( x_i \) the number of times case \( i \) applies. Then we have for all \( i \):

\[
\sum_i x_i = |C| \text{ with } l_j \cdot |C_j| \leq \sum_i x_i \cdot n_{ij} \leq u_j \cdot |C_j|, \text{ all } j \tag{15}
\]

### 4.2. Globally Bounding Component Numbers

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

If we have a binary connection axiom (1) with \( l_2 > 0 \) we include an edge from \( C_2 \) to \( C_1 \). For one-to-many axioms (4) and (5) 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 \( n_{ij} = 0 \) 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 \( \left( \bigwedge_i C_i \right) \Rightarrow C_1 \).

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

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

Thus, we have proven the following:

**Proposition 4.1 (Finiteness of configurations).** *It can be decided in linear time whether a given configuration problem is finite.*

Observe that this is a stronger result than the one presented in [Aschinger et al. 2011]: 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 \( S \) of vertices such that for every vertex there is a path ending in some vertex in \( S \) or the vertex is in \( S \) 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 \( O(N) \); 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 \( \Phi \) 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 \( \text{ilist} \), 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 \( \text{ilist} \) and are done. Otherwise we call a recursive procedure \( \text{enum} \). 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 \( \Phi \). Mark \( x_1 \) with 1 and put it on \( \text{ilist} \), 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 \( \text{ilist} \), otherwise recursively call \( \text{enum} \). (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 \( \text{enum} \). (Note that since \( x_1 \) is marked the number of unmarked variables has decreased, but is still nonempty.)

Note that every time, \( \text{enum} \) 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 \( \text{ilist} \) input components, all components corresponding to variables marked by 1 will be finite.

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

**Proposition 4.2 (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 4.3 (Cardinality-Minimal Sets of Inputs).** The problem to decide whether there is a set of components of size at most \( k \) that suffice to make the configuration problem finite is NP-complete.

**Proof Sketch.** The problem to decide if there is a key of size at most a given integer for a database under functional dependencies is NP-complete [Lucchesi and Osborn 1978]. A subset \( K \) of the database attributes \( A \) is a key if \( K \) and the functional dependencies determine all of \( A \).

Logically this problem can be expressed as follows: The attributes \( A \) become atomic propositions...
A functional dependency \( C \rightarrow B \) becomes an implication \((\land C) \Rightarrow (\land B)\); i.e. it can be expressed as Horn clauses. This proves hardness of our problem. Membership in \( \text{NP} \) follows by the straightforward approach to guess and verify a set of \( k \) input components that make the problem finite.

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

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

### 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 (12), (13), (14) and (15) can naturally be expressed in integer programming. Hence lower and upper bounds can be computed by solving two integer programs per generated component. On the other hand, we can reduce e.g. the subset sum problem to a LoCo problem giving rise to condition (15) and we have:

**Proposition 4.4 (Bounds computation is \( \text{NP-hard} \)).** Computing lower and upper bounds on the number of components needed to solve a configuration problem in LoCo is \( \text{NP-hard} \).

**Proof.** In the subset sum problem we are given a finite set \( A = \{1, \ldots, m\} \), a positive integer size \( s(a) \) for each \( a \in A \) and a positive integer \( B \) [Garey and Johnson 1979]. The problem is to determine whether there is \( A' \subseteq A \) such that \( \sum_{a \in A'} = B \). For the reduction all we need is a LoCo axiomatization containing a component type \( C_j \) for every \( a \in A \) and giving rise to condition (15)

\[
\sum_{i} x_i = |C| \text{ with } l_j \ast |C_j| \leq \sum_{i} x_i \ast n_{i,j} \leq u_j \ast |C_j|, \text{ all } j
\]

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

In [Feinerer 2013] it has been shown that computing lower bounds for problems containing only binary connections or basic one-to-many connections can be solved in polynomial time. The techniques used, however, do not extend to computing upper bounds or to one-to-many connection axioms of the form (15).

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

**Proposition 4.5 (tightness of bounds).** Assume given a LoCo axiomatization containing no constraints on the connection axioms, no partial configurations and no connection generating rules. Then the lower and upper bounds computed are tight if the integer programming solutions obtained satisfy the following additional conditions:

- \( |C_1| > 0 \Rightarrow |C_2| \geq l_1 \) for every binary connection axiom (1);
hold. However, as illustrated by the above example, for lower bounds this analogy does not
per bounds the maximum number of connections into a set is also an upper bound on that set's
are derived from the minimum and maximum number of connections between the respective sets
of a suitable size as well as suitable interconnections. We then observe that the linear inequalities
initial configurations and connection generating rules model finding reduces to finding component sets
of the form (4) the overall number of right hand side components is large enough; for an exclusive-
nevertheless be decided in NP. As the following result shows this only holds in the unlikely case of

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

So assume given a solution to the integer program. Then consider a binary connection axiom from
$C_1$ to $C_2$. The conditions $l_1 |C_1| \leq u_2 |C_2|$ and $l_2 |C_2| \leq u_1 |C_1|$, $|C_1| > 0 \Rightarrow |C_2| \geq l_1$ as well as
|C_2| > 0 \Rightarrow |C_1| \geq l_2$ jointly guarantee that we can find valid connections: For each $C_1$ there are at
least $l_1 C_2$ to connect it to and the overall number of $C_2$ is at least large enough to connect all the $C_1$
(and analogously in the other direction). In general, the linear “local bounds” inequalities guarantee
that there are enough components from the right hand side to connect all the components from the
component type on the left hand side of a connection axiom. The additional conditions satisfied
by the solution to the integer program guarantee that for each left hand side component there are
enough different right hand side components to connect it to: For a one-to-many connection axiom of the form (4) the overall number of right hand side components is large enough; for an exclusive-
or one-to-many connection axiom of the form (5) there is a sufficient number of at least one of the
right hand side components; and for a one-to-many connection axiom of the form (6) there are
sufficient numbers of the right hand side components so that at least one of the cases applies.

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

5. THE COMPLEXITY OF DECIDING LOCO SATISFIABILITY

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

PROPOSITION 5.1 (DECIDING LOCO SATISFIABILITY IS EXPTime-COMPLETE). Assume
given a LoCo domain axiomatization plus instance knowledge with an intensional component
catalogue. Then it is ExpTime-complete to decide whether there exists a model satisfying the
axioms.

PROOF. We first show membership in ExpTime. Deciding finiteness, identifying suitable sets
of input components and computing bounds on the number of components clearly can all be done
in ExpTime. We may hence assume that the number of instances for all component types is finitely
bounded. Observe that no component type may have more than exponentially many instances and
that all attributes have finite ranges. In particular, there are at most exponentially many different combinations of attribute values that satisfy the component catalogue. Hence in \textsc{Exptime} we can generate:

— for each component type \( C_i \) all possible component sets within the size bounds,
— all possible combinations of these sets, and
— all possible extensions of the \( C_i \rightarrow C_j \) relations.

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

Next we show hardness, by reducing an \textsc{Apspace} Turing machine to a \textsc{Loco} axiomatization; by [Chandra et al. 1981] this suffices to show \textsc{Exptime} hardness. The idea is to encode the machine configurations into components (all of the same type \( C_{TM} \)). Transitions are then encoded as connections between the components. We need to have \( \mathcal{O}(2^{p(n)}) \) components at our disposal. This is achieved by repeating the construction sketched immediately below proposition 4.4 using \( p(n) \) many component types and connection axioms. For each cell of the Turing machine we introduce an attribute. Hence the components are of the form \( C_{TM}(id, b, q, t_1, \ldots, t_{p(n)}) \). Here \( b \) denotes the position of the head and \( q \) indicates the machine state (including whether we are in an existential, universal, accepting or rejecting state). The \( t_i \) are the tape cells; the respective attribute values range over the tape alphabet. We may assume that the Turing machine terminates after exactly \( 2^{p(n)} \) steps and that component identifiers are numbered starting from “1”.

The initial state may then be encoded as \( C_{TM}(1, b, Q_0, \ldots, \ldots) \). We can now rule out rejecting configurations of the Turing machine by using \textsc{Loco}’s partial configurations: \( \forall \neg C_{TM}(id, b, reject, t_1, \ldots, t_{p(n)}) \). We still need to encode the machine’s transitions: For this we use a binary connection axiom of the form (2). We assume that in the universal states each configuration has exactly two successors whereas in the existential states there is exactly one successor.
\[(\forall id, b, q, \vec{t}) \ C_{TM}(id, b, q, t_1, \ldots, t_{p(n)}) \Rightarrow \]
\[
[ ( \exists_1^1 id C(id, b, \hat{q}, \vec{t}_1, \ldots, \vec{t}_{p(n)}) \land \phi_1 ) \lor \\
( \exists_2^2 id C(id, \hat{b}, \hat{q}, \vec{t}_1, \ldots, \vec{t}_{p(n)}) \land \phi_2 ) ] \\
\land \hat{id} = id + 1 ]
\]

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

\[\bigwedge_B [ (b = B \land t_b = T \land q = Q) \Rightarrow \\
(\hat{b} = \hat{B} \land \hat{t}_b = \hat{T} \land \hat{q} = \hat{Q} \land \bigwedge_{i \neq B} t_i = \hat{t}_i) ] \]

Given this basic setup it is straightforward to complete the definition of the LoCo axiomatization in such a way that the \(APSPACE\) Turing machine accepts if and only if the corresponding LoCo axiomatization has a model: We stipulate that there are no connection axioms beyond the binary connection from \(C_{TM}\) to itself just sketched and the binary connections used in the construction below proposition 4.4 in order to obtain \(2^{p(n)}\) instances of \(C_{TM}\). For the latter we may assume that the respective component types have no attributes beyond their identifiers and hence that there are no \(\phi\)-constraints in the connection axioms. Next, we may assume that the component catalogue neither specifies nor rules out any attribute combinations for \(C_{TM}\). Finally, the partial configuration used in the LoCo axiomatization shall consist only of the Turing machines initial state and the axiom ruling out rejecting configurations. \(\square\)

**Discussion**

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

**6. 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. As already mentioned the idea of LoCo is to serve as a source language that gets translated into different target languages. Next to the implemented translation to answer set programming (ASP) we have developed a translation into MINIZINC [Nethercote et al. 2007] on a theoretical level. MINIZINC is a medium-level constraint modelling language gaining a lot of interest recently with the aim of becoming a standard modelling language for the Constraint Programming community. A transformation to MINIZINC provides us with access to a whole portfolio of integer programming, SAT and constraint solvers.

The LoCo axiomatization is defined via the LoCo input language which is basically a text-based representation of it. A detailed description of the LoCo input language as well as the translation of
all the language elements is beyond the scope of this paper and, together with a thorough evaluation, subject of future work. However, we show the basic idea by a transformation of a binary connection in section 6.1, one of the cornerstones of our formalism, and also some benchmark results on our running example as well as on another industrial problem in section 6.2.

Figure 3 depicts an overview of the transformation workflow. First the problem gets specified in the LoCo input language; domain and instance knowledge can optionally be defined in separate input files. Our aim concerning future work is to develop a graphical user interface that creates the textual LoCo representation automatically. Next we parse the input files including a syntax and semantic check for correctness. The system is able to give the user appropriate and practical feedback in case of any errors. After that we determine bounds for the generated components and check if the model is finite by using a configuration graph constructed during the parsing step. The specification is then translated into executable code, resulting in one or more output files depending on the chosen target language and the chosen output mode. For example, the ASP implementation allows to translate instance and domain knowledge in separate files or altogether in one output file.

We have prototypically implemented LoCo in ASP. The motivation for this choice was twofold: (1) Being a dialect of rule-based logic programming [Gelfond 2008] ASP allows a natural representation of LoCo problems, especially when extended with so-called cardinality constraints (see below). (2) In our implementation we use the Potassco framework [Gebser et al. 2011] which gives us access to a state-of-the-art conflict-driven clause learning solver much like contemporary SAT solvers. This technology has proven to be both highly efficient and robust on numerous academic and industrial challenge problems.

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

### 6.1. Transformation to ASP

In this section we will transform the connection from thing to cabinet as already shown in Example 3.1 together with the reverse direction shown in Example 6.1 below.

**Example 6.1.** In the House Problem each cabinet contains between one and \( cMax \) things of type \( A \); moreover, for each cabinet the sum of the size of all big things is not allowed to be greater than five:

\[
(\forall id_C, cSize, cBig, cTop) \text{ cab}(id_C, cSize, cBig, cTop) \Rightarrow \\
(\exists 1^{cMax} id_TA) \text{ thingA2Cab}(id_TA, id_C) \land \text{ thingA}(id_TA, tSize, tBig) \land \\
\left[ \sum (tSize \ [tBig = \text{TRUE}]) \leq 5 \right]
\]
Aggregate functions can have optional nested constraints. The $\sum$ aggregate in the constraint part calculates the total size of all things in the cabinet, restricted to those things with attribute big set to TRUE by a nested constraint inside the aggregate function.

1{thingA_Cab(T,C):cabGen(C)}1 :- thingA(T).
1{thingA_Cab(T,C):thingAGen(T)}cMax :- cab(C).
:- thingA2Cab(T,C), not cab(C).
:- thingA2Cab(T,C), not thingA(T).

% constraint thingA -> cab
:- not c1(T,C), thingA2Cab(T,C).
c1(T,C) :- thingA2Cab(T,C), tBig(T,TB), cBig(T,CB), TB==CB.
c1(T,C) :- thingA2Cab(T,C), tBig(T,B), B==0.

% constraint cab -> thingA
:- not c2(T,C), thingA_cab(T,C).
c2_Aggr1(C,T,TS) :- thingA_cab(T,C), tSize(T,TS), c2_Constr1(T,C).
c2_Constr1(T,C) :- thingA_cab(T,C), tBig(T,TB), TB == 1.
c2(T,C) :- thingA_cab(T,C), TS_SUM = #sum[c2_Aggr1(C,_,TS) = TS], TS_SUM <= 2.

Listing 1: ASP output for a binary connection

The code snippet in listing 1 shows the transformation of the binary connection in both directions. The first line represents the connection from thing to cabinet as shown in Example 3.1 while the second represents the reverse direction from cabinet to thing. Both lines use so-called cardinality constraints [Simons et al. 2002] with the numbers or constants outside the curly brackets specifying the lower and upper bounds: Line 1 expresses the fact 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 conditional part (cabGen(C)) in such rules must be specified by ground facts in the knowledge base. Hence our knowledge base contains all possible instances of thingAGen and cabGen, i.e. the finitely many component instances which might be used in the configuration. The instances of cab and thingA are those that actually are used as part of a solution set. The integrity constraint in line 3 ensures that every cabinet that features in a connection is also in the extension of the cab predicate. The same is done analogously for thing of type A in line 4.

Lines 7-9 depict the mapping of the constraint subformula from Example 3.1. The integrity constraint in line 7 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 and cabinet have the same value for attribute big (line 8) or thing is not big (line 9), expressing the fact that big things can only be put in big cabinets.

Lines 12-16 show the transformation of the constraint subformula from Example 6.1. Line 12 depicts an integrity constraint for constraint $c2$ analogous to line 7. The main constraint is shown in lines 15-16, ensuring that the calculated sum $TS\_SUM$ is at most two. $TS\_SUM$ represents the sum of all thing sizes for a certain cabinet C in extension of predicate $c2\_Aggr1$ (line 13) which represents the combination of the connection predicate with the size attribute of thing. However $c2\_Aggr1$ only contains those tuples that satisfy the nested constraint $c2\_Constr1$ (line 14) ensuring that only big things will be taken into consideration. The structure of a nested constraint is analogous to a standard constraint.

Using similar transformation steps we are able to correctly map arbitrary boolean combinations of arithmetic expressions, attribute comparisons and aggregate functions to ASP. 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 (as given by the derived finite upper bound); as already pointed out above this may be exponentially many in the size of the domain axiomatization. In the solving process LoCo then picks a subset of those generated components, possibly a minimal subset in an optimization problem, which form valid models satisfying the specification. The idea to start from lower bound many components only to incrementally add components until a configuration is found is practically very appealing. This would resemble the basic approach of generative CSP with additional upper bounds. However, there are currently no freely available solvers for generative CSP which we could use as a potential LoCo target language. Also the existing tool support concerning ASP does not yet meet our needs for implementing this idea [Gebser et al. 2008]. For solving optimization problems, in our implementation we support expressing cost functions subject to minimization and/or maximization that can be used after assigning weights to components or connections. On the theoretical side the issue of optimization in LoCo still poses interesting questions for future work.

An important point that needs to be addressed is the fact that the source and target languages may have different semantics in terms of negation. In particular, the LoCo axiomatization uses classical (strong) negation whereas ASP was conceived using default negation [Gelfond and Lifschitz 1991] and only later extended to support classical negation, too. Default negation basically expresses the fact that a literal \textit{not} \textit{L} holds by default unless \textit{L} is derived whereas the classical negation of \textit{L} holds only if the complement of the proposition, expressed by \textit{¬L}, can be derived. Our approach for solving this is to use classical negation throughout the whole workflow. However, LoCo axiomatizations allow the negation of arbitrarily complex constructs e.g. in the \textit{φ} constraint subformulas of rules. This could involve an arithmetic expression or a combination of several expressions by logical connectives whereas classical negation in ASP can be applied to atoms only. We sidestep this issue by transforming every \textit{φ} formula into negation normal form (NNF) such that negation occurs only at the atomic level. If the respective atom is an attribute comparison, a component or a connection atom then we can directly use classical negation in ASP. In case the atom is an arithmetic expression we eliminate negation by transforming the expression, e.g. by changing \textit{¬(3a + 5b ≤ c)} to \textit{3a + 5b > c} or as in line 9 for a boolean attribute \textit{big} by changing \textit{true} to \textit{false} and vice versa.

Note also that our translation does not rely upon disjunctive ASP dialects, i.e. ASP extensions for modelling inclusive/exclusive disjunctions. A well-known problem with the ASP stable model semantics is that inclusive disjunction could be falsely interpreted as exclusive disjunction because of the minimal model interpretation. By using cardinality rules with appropriate bounds instead we are able to enforce the intended disjunction semantics in (exclusive-or) one-to-many connections. Disjunctions in the \textit{φ} constraint subformulas are generally inclusive, for an example of how this is treated see lines 8-9.

### 6.2. Benchmarks

We have evaluated our encodings on a set of benchmark instances that we received from our industrial partners.\footnote{Available from: http://proserver3-iwas.uni-klu.ac.at/reconcile/index.php/benchmarks} We compare the runtimes of our LoCo implementation against previously developed hand-crafted ASP encodings. The experimental results that we obtain are very encouraging: For the House Problem we can compete with the hand-written problem encoding in answer set programming\footnote{Presented in [Friedrich et al. 2011]; it turns out that our translation of the declarative LoCo specification yields a very similar program.} presented in [Friedrich et al. 2011]; it turns out that our translation of the declarative LoCo specification yields a very similar program.

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

<table>
<thead>
<tr>
<th>Problem instance</th>
<th>Search</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LoCo encoding</td>
<td>Manual encoding</td>
</tr>
<tr>
<td>p02t06</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>p02t10</td>
<td>0.05</td>
<td>0.03</td>
</tr>
<tr>
<td>p03t15</td>
<td>0.07</td>
<td>0.04</td>
</tr>
<tr>
<td>p04t20</td>
<td>0.13</td>
<td>0.05</td>
</tr>
<tr>
<td>p05t25</td>
<td>0.18</td>
<td>0.06</td>
</tr>
<tr>
<td>p10t50</td>
<td>0.92</td>
<td>0.21</td>
</tr>
<tr>
<td>p15t75</td>
<td>2.95</td>
<td>0.51</td>
</tr>
<tr>
<td>p20t100</td>
<td>7.14</td>
<td>1.19</td>
</tr>
<tr>
<td>p30t150</td>
<td>24.72</td>
<td>4.35</td>
</tr>
<tr>
<td>p40t200</td>
<td>65.15</td>
<td>25.92</td>
</tr>
</tbody>
</table>

We have evaluated the instances in two different settings: (1) the original setting (Optimization) and (2) a simplified version (Search) where the effectively needed number of all components is already part of the input. In the latter case the search for the minimal number of needed components is eliminated and the configuration problem reduces to connecting components correctly with regard to the side constraints. In general the outcome of our experiments can be summarized as follows: The LoCo encoding performs slightly worse than the manual encoding in both settings. This overhead probably has to be attributed to the additional helper predicates created when transforming the constraint parts of the rules. In the easier “Search” setting all instances are solvable whereas in the “Optimization” setting both the LoCo and the manual encoding run into timeouts for the same instances.

Table II: Benchmarks for the Partner Units Problem

<table>
<thead>
<tr>
<th>Problem instance</th>
<th>Search</th>
<th>Optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LoCo encoding</td>
<td>Manual encoding</td>
</tr>
<tr>
<td>small-7</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>small-8</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>single-11</td>
<td>0.15</td>
<td>0.1</td>
</tr>
<tr>
<td>small-no</td>
<td>1.18</td>
<td>0.95</td>
</tr>
<tr>
<td>double-10</td>
<td>0.15</td>
<td>0.08</td>
</tr>
<tr>
<td>double-14</td>
<td>0.77</td>
<td>0.19</td>
</tr>
<tr>
<td>double-16</td>
<td>1.32</td>
<td>0.53</td>
</tr>
<tr>
<td>double-20</td>
<td>44.15</td>
<td>1.52</td>
</tr>
</tbody>
</table>

Another practical problem we received from our industrial partners is the Partner Units Problem (PUP) [Falkner et al. 2011]. The problem originated from configuring railway interlocking systems but has also widespread practical relevance in other domains such as security and surveillance systems. It has recently also been introduced as a benchmark problem in the Third Answer Set Programming Competition [ASP Competition 2011] where it turned out to be one of the most difficult problems to solve efficiently. We have evaluated the LoCo version of the PUP and were able to reach about the same performance as the answer set program presented in [Aschinger et al. 2011] if for the latter the problem-specific search strategy is turned off (see table II for results). In particular, the gap between the automatic translation and the hand-written problem encoding is similar to what we have experienced for the House Problem which further justifies our trust in the quality of the automated LoCo transformations. Aschinger et al. [2011] further compares the ASP version with SAT, CSP and integer programming encodings as well as with a Java-based implementation optimized for a special case. Since all these encodings are using problem-specific solving heuristics we are not taking them into account for the benchmarks presented here.
7. CONCLUSION

We conclude the paper by summarising the main points of LoCo and by pointing out promising directions for future research.

7.1. Summary of LoCo

The main scientific contribution of LoCo as a whole to the area of configuration research can be summarized as follows: Like conditional or generative CSP LoCo supports the conditional inclusion of components into configurations as a knowledge representation idiom. In contrast to competing approaches, in LoCo the number of available components does not have to be specified manually for all component types involved; yet LoCo does not face termination issues. Apart from this, LoCo features ports, a means to describe arbitrary component connection layouts as well as a rich language for describing constraints on the admissible combinations. Moreover, the user can specify partial configurations (not) to be used for building the configuration. Our prototypic implementation in answer set programming has proven that our approach is applicable in practice.

7.2. Future Work

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

In order to increase the practical usability of LoCo it would be helpful to either develop a graphical user interface or to single out a fragment/extension of UML and OCL corresponding to LoCo.

On the reasoning side there also remain a number of challenges: On the one hand side, the implementation via answer set programming is now mature, and a detailed description thereof together with a thorough experimental evaluation is currently underway. On the other hand, we would like to have complementary reasoning methods at our disposal. To this end we have theoretically developed a translation into the MINIZINC language [Nethercote et al. 2007], providing us with access to a whole portfolio of integer programming, SAT and constraint solvers; however, an implementation is still missing. It would also be promising to develop an implementation that follows the idea of incrementally adding components instead of pre-generating all potentially useful ones; here, the absence of freely available solvers for generative or conditional CSP constitutes an additional obstacle.

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

REFERENCES


LoCo — A Logic for Configuration Problems


