Introducing LoCo, a Logic for Configuration Problems[*]

Markus Aschinger, Conrad Drescher, Georg Gottlob

Department of Computer Science, University of Oxford

`firstname.lastname@cs.ox.ac.uk`

In this paper we present the core of LoCo, a logic-based high-level representation language for expressing configuration problems. LoCo shall allow to model these problems in an intuitive and declarative way, the dynamic aspects of configuration notwithstanding. Our logic enforces that configurations contain only finitely many components and reasoning can be reduced to the task of model construction.

1 Configuration Problems

Configuration systems are one of the most successful applications of AI-techniques. In industrial environments, they support the configuration of complex products and, compared to manual processes, help to reduce error rates and increase throughput [12]. The following definition by Mittal and Frayman [10] describes what is typically meant by a configuration problem.

Definition 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.*

    In typical configuration problems, the number of components needed for a solution is unknown beforehand; for example, for some components this number depends on the choices made for other components. One can think of this as of creating new components on-the-fly throughout the solving process. Existing knowledge representation (KR) tools able to express this dynamic aspect of configuration require that explicit bounds on all generated components be given as well as extensive knowledge about the underlying solving algorithms.

    In this work we introduce a purely declarative logical formalism where the KR engineer only has to specify the possible numbers of connections between any two component kinds. From this information finite bounds on the number of components needed in a configuration are inferred — that is, in any model of the configuration problem the number of components used is finite. Formally this logic is a fragment of classical First Order Logic (FO), extended by existential counting quantifiers. We plan to eventually develop translations from the logic representation into a low-level input format for various solvers, e.g. SAT or Integer Programming.

---

2 Configuration Formalisms

Over the years several different approaches for configuration have been investigated, e.g. expert systems, rule-based systems, non-monotonic reasoning, case-based reasoning, description logics and constraint processing. A recent survey is given by Junker in [6].

2.1 Constraint-Based Formalisms

Constraint satisfaction problems (CSPs) are currently the most widely used approach for the formalisation of configuration problems. However, the standard CSP formulation does not feature variables or sub-CSPs that are conditionally activated depending upon the values assigned to other variables.

Hence, in the area of constraint-based configuration, a number of extensions of the traditional CSP paradigm have been developed. In Conditional CSPs [9] activation constraints ensure that only a relevant subset of the variables and constraints is used for generating a solution. In Composite CSPs [11] variables can have subproblems (sub-CSPs) as values. In both formalisms the number of possibly activated variables and constraints has to be defined in advance. Accordingly, both formalisms admit translations into classic CSPs [14].

A Generative CSP (GCSP) [13] allows the dynamic generation of components on demand during the search process. The reasoning starts from certain key components and then required auxiliary components and associated connections are incrementally added. No explicit bounds on the number of components have to be given and the formalism allows infinite configurations to be constructed.

2.2 Logical Frameworks

There have also been some previous attempts to capture configuration with logic-based formalisms. We recall these in some detail, as they are the starting points for our configuration logic.

Classical CSPs correspond to the fragment $\exists FO_{\wedge,+}$ of $FO_{\wedge,+}$ of FO consisting of formulae built using only existential quantification and conjunction [7]:

Definition 2 *The logical counterpart of a CSP is defined as a pair $(\phi, \mathcal{D})$, where $\mathcal{D}$ is the constraint database, i.e., the extensional representation of all the constraint relations and ϕ is a $\exists FO_{\wedge,+}$ sentence. Solving the CSP corresponds to deciding whether $\mathcal{D} \vDash \phi$.*

In the work by Gottlob et al. [5] logical implication has been added to this formalism to express the conditional inclusion of components into configurations. This $\exists FO_{\rightarrow,\wedge,+}$ fragment of FO is one of the starting points for our own formalism. For example, it allows us to ask whether $\mathcal{D} \vDash (\exists x) \mathrm{Car}(x) \wedge (\mathrm{LuxuryCar}(x) \Rightarrow \mathrm{HasSunRoof}(x))$. A drawback of $\exists FO_{\rightarrow,\wedge,+}$ is that explicit bounds on the number of components needed has to be given (variables have a fixed finite domain) and that all constraints must be coded in extension in the constraint database.

There are also two prominent formalisms based on Description Logics (DLs): The works by McGuinness et al. [8] and Klein et al. [1]. These are the other two starting points of our formalism.

In both works valid configurations are described using DL axioms. DLs are fragments of FO based on unary and binary predicates, so-called concepts and roles. In both approaches concepts are used for describing components and attributes; roles are used to describe the relations between components and also between components and attributes. Klein et al. reduce the task of finding a valid configuration to the problem of constructing a *finite* model of the axioms. McGuinness et al. propose an interactive approach where (1) the knowledge engineer adds atomic propositions to the

axioms and (2) the inference engine computes the consequences until (3) eventually a finite model is obtained. The DLs from both formalisms always admit both finite and infinite models; hence no explicit bound on the number of components has to be given. The absence of predicates of arity greater than two can make domain encodings unnecessarily complex.

Finally, in [4] a logic-based formulation of GCSPs has been given; this formulation does not require that bounds on the component numbers be given, but admits infinite configuration models.

3   The LoCo **Formalism**

We now introduce the core of LoCo, a new logic-based framework for modelling practical configuration problems. In this work we do not yet address ports or optimal configurations. The basic idea is to describe a configuration problem (the problem domain) by a set of logical sentences. The task of finding a configuration is then reduced to the problem of finding a model for the logical sentences — this is the same approach as the one taken by Klein et al. [1]. From Gottlob et al. we take the idea to express the conditional existence of components in configurations via implication and existential quantifiers. However, we use counting quantifiers for this, and these are already present in the work by McGuinness et al. (albeit used for a different purpose). The main idea of LoCo is that via these counting quantifiers we can enforce that each model of the configuration problem contains finitely many components only.

3.1   Formal Basics

Formally, LoCo is based on a fragment of classical logic with equality interpreted as identity. This fragment is then extended with existential counting quantifiers.

Components: Components are modelled as n-ary predicates $Component(id, \vec{x})$, with id the component's identifier, and $\vec{x}$ a vector of component attributes. Components are of various kinds; we will denote individual kinds by C_1, C_2.

Typed Variables: It is convenient to say that the different arguments of components have different types. We will introduce one type ID for each identifier of a component kind and also for each attribute type. We assume that there are only finitely many different types in the configuration domain that are all mutually disjoint. In our notation we will use typed variables in formulas.

We now show how these typed variables can be accommodated in classical first order logic — - this is very similar to the reduction of many-sorted logic to classical FO (cf. e.g. [2]). We first introduce unary predicates for each type (e.g. ID for type ID) and add domain partitioning axioms:

$$(\forall x) \bigvee_{T \in \mathcal{TYPES}} T(x),$$

$$(\forall x) \bigwedge_{T_i, T_j \in \mathcal{TYPES}, i \neq j} \neg(T_i(x) \wedge T_j(x)).$$

Then for transforming a typed formula to an untyped one we replace e.g. each subformula $(\forall id)\phi(id)$ by $(\forall x)\text{ID}(x) \Rightarrow \phi(x)$ and likewise $(\exists id)\phi(id)$ by $(\exists x)\text{ID}(x) \wedge \phi(x)$ — this is the standard reduction from many-sorted to classical FO. However, for the moment we are not going to introduce types for the terms (other than variables) of the language. Later we are going to stipulate that there are *standard names* for the elements in the domain of each type, cf. Section 3.2.

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, we might have a formula $\exists_l^u x \phi(x)$ enforcing that the number of different x (here x denotes a sequence of variables) such that $\phi(x)$ is restricted to be within the range $[l, u]$. In classical logic without counting quantifiers this can be expressed as

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

As usual quantifiers range over a single type 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 types. 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. We denote exclusive disjunction between types in these subformulas by $\exists i_l^u x \phi(x) \vee \psi(x)$ — this abbreviates a formula enforcing that the total number of objects such that ϕ is between l and u and there are no x such that ψ (or the other way around).

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 $C_2 2 C_1$ - it does not matter which. We allow connections from a component type to itself, i.e., $C2C$. A predicate $C_i 2 C_j$ is of type $\text{ID}_i \times \text{ID}_j$. For every connection predicate $C_1 2 C_2$ two formulas are included:[1]

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

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

The first formula says how many components of kind C_2 can be connected to any given component of kind C_1, with the subformula ϕ (with variables among $id_1, id_2, \vec{x}, \vec{y}$) expressing additional constraints, like e.g. an aggregate function $\sum n \leq Capacity$. The second formula is for the other direction. If the connection is from a component kind to itself only one of the formulas is included.

The formulas ϕ and ψ for expressing constraints on the connections consist of conjunctions and disjunctions of linear arithmetic expressions and (in-)equalities between terms. We believe this to be sufficient for many practical examples; if necessary we will broaden the language, but we have to keep in mind the planned translation to executable formats.[2]

Next to the rules for binary connections, there are also rules for supporting one-to-many connections (3), i.e. connecting one component with a set of components. For every one-to-many connection the component on the left-hand side needs to have binary connections to all components in the set on the right-hand side. This is mandatory for the propagation of bounds and will be discussed later on. Note also that the single component is not allowed to be part of the set.

$$(\forall id, \vec{x}) C(id, \vec{x}) \Rightarrow (\exists_l^u id_i) \left[\bigvee_i C 2 C_i(id, id_i) \wedge C_i(id_i, \vec{y}) \right] \tag{3}$$

---

[1] Throughout this paper free variables in formulas are to be read as existentially quantified from the outside.

[2] For an explicit model of ports in LoCo we can introduce attribute types for the ports and a binary predicate ConnectionPorts that is then used in ϕ. We do not do so here in order to simplify the presentation.

In this rule the quantifier $\exists_l^u$ ranges over the $i > 1$ different ID types. It may also be replaced by the $\exists i_l^u$ quantifier enforcing that each C_1 is connected to components of only one of the C_i kinds. The cardinality upper bound is optional and in combination with the binary connections, a sufficient bound can be automatically computed.

3.2   Specifying Configuration Problems

The specification CP 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 for the former we know exactly how many are used in a configuration and for the latter we don't. We stipulate that a configuration problem always includes at least one component of the input variant.

3.2.1   Domain Knowledge

Connection Axioms Connection axioms take the form introduced above. Only in binary connection rules we allow the lower bound to be zero in the $\exists_l^u$ quantifier, i.e. we can have $l = 0$. Without further conditions this would allow us to include infinitely many components into configurations: Assume we have two components C_1 and C_2, where each C_1 is connected to exactly one C_2, and each C_2 is connected to at most one C_1. It does not help if we know exactly how many C_1 there are (say n): Still we can have infinitely many C_2 that are not connected to any of the C_1.

We address this problem as follows: First, the component kinds have to be divided into the classes *input*, *generated* and *both*. Then we stipulate that for every rule for binary connections from C_1 to C_2 with a lower bound of zero: C_1 is *input*, or there is some other binary or one-to-many connection from C_1 with lower bound greater than zero. Then we define a level mapping on the component kinds via the connection axioms: Input components are on level zero. On level one are those generated components for which there is a (binary or one-to-many) connection axiom with lower bound greater than zero from the component to only input components. Level two components are grounded in input or level one components, and so on.

Now any domain knowledge axiomatisation has to fulfil the following property: No matter how the subdivision of component kinds into the classes *input*, *generated* and *both* is instantiated there has to exist a level mapping of the components such that all components of the *generated* variety are assigned to some finite level. The existence of such a level mapping can be checked by first assigning *generated* to the components belonging to the class *both* and then doing a graph traversal starting from the input components.

Attribute Naming For all attribute types a naming-scheme is included. For ordinary component attributes these take the form (4) where T is the unary type-predicate for the given type and $\mathcal{V}$ is a finite set of ground terms, the possible attribute values:

$$(\forall x)\, T(x) \equiv \bigvee_{V \in \mathcal{V}} x = V \tag{4}$$

For component attributes of type ID the naming-scheme has the form (5) where $\phi(x)$ is a FO formulation of the (infinitely many) possible names of elements in that type. For example, this could be a simple numbering axiom of form: $(\forall x)S(x) \Rightarrow (\exists n)x = \text{SName}(n)$.

$$(\forall x)\, ID(x) \Rightarrow \phi(x), \tag{5}$$

By default, unique name axioms for all distinct terms are also included. Hence naming-scheme axioms of the form (4) force the domain of the type to be equal to the set of all terms t such that $\phi(t)$, whereas the form (5) only forces the domain of the type to be a subset thereof.

To sum up, for each component kind the ID attribute is unbounded, but ordinary attributes can have only finitely many distinct values. However, in each model of a configuration problem only finitely many components will exist. We introduce a new variable type EXCESS without naming-scheme axiom: The names of components not used in a configuration can be discarded by assigning them to this type. Finally, for every component kind we introduce an axiom

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

expressing the fact that, in database terminology, the respective ID is a *key*.

Component Catalogue For each component kind the so-called catalogue contains information on the instances that actually can be manufactured. We express this as axioms (where each $\vec{V_i}$ is a tuple of ground attribute values):

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

3.2.2 Instance Knowledge

On the instance level the components assigned to the class *both* have to be divided into input and generated components. 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 ID_i and ID is the respective type predicate. This axiom is stronger than the naming-scheme for the component; hence, in any model identifiers mentioned in the naming-scheme axiom but not in the domain closure axiom will belong to the type EXCESS.

Both input and generated components that have to be used in the configuration can be explicitly listed as atoms (with possibly uninstantiated, existentially quantified arguments). We also allow positive and negative ground connection predicates like, for example, $\neg C_1 2 C_2(ID_1, ID_2)$.

3.3 Finite Model Property

Next we are going to show that in any model of a configuration domain specification for all components the domain of the ID attribute is finite.

Proposition 1 (Configurations contain finitely many components) *Let* CP *be a configuration domain specification and* $\mathcal{I}$ *be an interpretation such that* $\mathcal{I} \models$ CP. *Then for all components the domain of the* ID *attribute is finite in* $\mathcal{I}$.

Proof Sketch: Assume that for a component C the domain of the ID attribute is infinite and that C is connected to some other component(s) C_i via a binary or one-to-many connection such that the domain of all the C_i is finite in $\mathcal{I}$. Then $\mathcal{I}$ is not a model for CP. The existence of a level mapping guarantees that each component is grounded in components with finite domains. $\dashv$

Calculating upper and lower bounds: In order to be able to transform a problem model into e.g. SAT or OPL, we need to know the lower and upper bounds on the number of instances for each component of the "generated" variety. For computing these possible domain sizes of generated components, we extract Diophantine inequalities from the connection formulas. This builds up on the work by Falkner et al. about semantics of UML class diagrams and cardinalities applied to the configuration domain [3].

Assume a binary connection defined by formulas (1) and (2), where C_1 is an input and C_2 is a generated component. We can calculate upper and lower bounds for component C_2 as follows:

$$l_1 * \left|C_1\right| \leq n \leq u_1 * \left|C_1\right| \tag{6}$$
$$l_2 * \left|C_2\right| \leq n \leq u_2 * \left|C_2\right|$$

$$l_1 * \left|C_1\right| \leq u_2 * \left|C_2\right| \tag{7}$$
$$l_2 * \left|C_2\right| \leq u_1 * \left|C_1\right|$$

The number of possible links n between the components is bounded as shown in (6). From this we can derive inequalities representing the relation between C_1 and C_2 (7). After some simple combinatorics we get lower bound $LB = \left\lceil \frac{l_1 * |C_1|}{u_2} \right\rceil$ and upper bound $UB = \left\lfloor \frac{u_1 * |C_1|}{l_2} \right\rfloor$, resulting in formula (8) for the bounds of C_2. It can be seen from the formula that to define a lower resp. an upper bound for C_2, we need the cardinality bounds l_2 resp. u_2 in the direction of C_1. The described computation also applies to connections between two generated components, provided that component C_1 has properly defined bounds. In this scenario we insert the lower bound on C_1 for computing LB and the upper bound on C_1 for computing UB of C_2.

$$\left\lceil \frac{l_1 * \left\lfloor |C_1| \right\rfloor}{u_2} \right\rceil \leq \left|C_2\right| \leq \left\lfloor \frac{u_1 * \left\lceil |C_1| \right\rceil}{l_2} \right\rfloor \tag{8}$$

$$\left\lceil \frac{\sum_i l_i * \left\lfloor |C_i| \right\rfloor}{u} \right\rceil \leq |C| \leq \left\lfloor \frac{\sum_i u_i * \left\lceil |C_i| \right\rceil}{l} \right\rfloor \tag{9}$$

In the case of one-to-many connections as shown in formula 3 new bounds are calculated for the component on the left-hand side. For this computation we combine a one-to-many connection with all existing binary connections between the current component and the components on the many-side. In other words, we take the cardinalities from a one-to-many constraint in direction to the set and the cardinalities of the binary connections in direction to the current component and compute bounds analogously to a simple binary connection (see formula 9).

The above procedures refine the bounds on the domain of components in single connections. However, if the domain size of one component is updated, then the domain size of other components may have to be updated again. We also have to take into account the one-to-many connections. The algorithm introduced below addresses both tasks until eventually for the domain sizes a fixpoint is obtained — or a contradiction has been detected.

First, the algorithm puts all input components on a stack in any order (line 2). The algorithm then takes a component off the stack and iteratively determines all binary connections including the current component (line 5). We perform a bound computation for each binary connection with a generated component and check if new bounds were computed for this generated component (line 8). If this is the case, we update the bounds, taking the maximum of all computed lower bounds and the minimum of all computed upper bounds. After an update we check the bounds for consistency, i.e. we check that lower bound ≤ upper bound (line 10) and put the connected component on the stack for further propagation of bounds (line 11). The algorithm terminates with an error whenever bounds become inconsistent.

If the current component is of type generated, then we also check if there exist one-to-many connections to a set of other components and iterate over them (line 14). In the case of one-to-many connections new bounds are calculated for the current component and not for the connected components. If we obtain new bounds for the current component, we perform an update and a consistency check similar to what is done for the binary connections and put the component back on the stack again to propagate the new bounds via the binary connections.

The algorithm then iteratively pops the next component off the stack and does the same computation step until the stack is empty. The algorithm is guaranteed to terminate and ensures the proper computation of maximal lower and minimal upper bounds on all generated components.

BOUND-PROPAGATION

```
 1   create an empty stack
 2   put all inpComp ∈ INPUT-COMPONENTS on the stack in any order
 3   while stack is not empty
 4       do currComp ← POP(stack)
 5           for all ⟨currComp,nbComp⟩ ∈ BINARY-CONNECTIONS
 6               do if nbComp ∈ GENERATED-COMPONENTS
 7                   then COMPUTE-BOUNDS(currComp,nbComp)
 8                       if NEW-BOUNDS(nbComp)
 9                           then UPDATE-BOUNDS(nbComp)
10                               if LB(nbComp) ≤ UB(nbComp)
11                                   then PUSH(nbComp,stack)
12                                   else  REJECT
13           if currComp ∈ GENERATED-COMPONENTS
14               then for all ⟨currComp,nbComps⟩ ∈ 1-TO-MANY-CONNECTIONS
15                       do COMPUTE-BOUNDS(currComp,nbComps)
16                           if NEW-BOUNDS(currComp)
17                               then UPDATE-BOUNDS(currComp)
18                                   if LB(currComp) ≤ UB(currComp)
19                                       then PUSH(currComp,stack)
20                                       else  REJECT
21   ACCEPT
```

4 Example: Modified Bin-Packing

We want to explain our approach by means of a simple Bin-Packing example, where we distinguish between two component kinds of *Things A* and *B* with all *Things* having a certain size. The *Bins* have an upper bound on how many *Things* of each kind can be put into them. *Things* are input components while the *Bins* are generated components with the aim of their number being minimised. The problem can be described by the following formulas:

$$\forall(id_{\text{TA}}, size)\, C_{\text{TA}}(id_{\text{TA}}, size) \Rightarrow \tag{10}$$
$$(\exists_1^1\, id_{Bin})\, C_{\text{TA}}2C_{Bin}(id_{\text{TA}}, id_{Bin}) \wedge C_{Bin}(id_{Bin})$$

$$\forall(id_{Bin})\, C_{Bin}(id_{Bin}) \Rightarrow \tag{11}$$
$$(\exists_0^5\, id_{\text{TA}})\, C_{\text{TA}}2C_{Bin}(id_{\text{TA}}, id_{Bin}) \wedge C_{\text{TA}}(id_{\text{TA}}, size) \wedge \sum size \leq 5$$

$$\forall(id_{\text{TB}}, size)\, C_{\text{TB}}(id_{\text{TB}}, size) \Rightarrow \tag{12}$$
$$(\exists_1^1\, id_{Bin})\, C_{\text{TB}}2C_{Bin}(id_{\text{TB}}, id_{Bin}) \wedge C_{Bin}(id_{Bin})$$

$$\forall(id_{Bin})\, C_{Bin}(id_{Bin}) \Rightarrow \tag{13}$$
$$(\exists_0^2\, id_{\text{TB}})\, C_{\text{TB}}2C_{Bin}(id_{\text{TB}}, id_{Bin}) \wedge C_{\text{TB}}(id_{\text{TB}}, size) \wedge \sum size \leq 2$$

$$\forall(id_{Bin})\, C_{Bin}(id_{Bin}) \Rightarrow \tag{14}$$
$$(\exists_1\, id_T)\, (C_{\text{TA}}2C_{Bin}(id_T, id_{Bin}) \wedge C_{\text{TA}}(id_T, \vec{y}))\, \vee$$
$$(C_{\text{TB}}2C_{Bin}(id_T, id_{Bin}) \wedge C_{\text{TB}}(id_T, \vec{y}))$$

Formula (10) states that every *ThingA* has to be put into exactly one *Bin*. The backwards-direction in formula (11) determines that a *Bin* has a total size bound of 5 for *ThingA*. Up to 5 of those things can be put into a *Bin* in case all those *Things* have minimum size 1 (hence the cardinality upper bound is 5). Formulas (12) and (13) analogously define the binary connection for *ThingB*.

Assume having an instance with 20 *Things* of each kind, connection *ThingA-Bin* gives a lower bound of 4 and connection *ThingB-Bin* gives a lower bound of 10 for component *Bin* using the bound computations defined in (8). We take the maximum of all available values, hence the lower bound for *Bin* is 10. Notice that in (11) and (13) the cardinality lower bounds of the connections are defined as zero to express the situation that a *Bin* could contain only one kind of *Thing* without the other. This results in the fact that we can't compute an upper bound for *Bin* using the binary connections defined so far and this would violate the finite model requirement. In order to express that for a *Bin* to exist it needs to have at least one *Thing* in it, we define a one-to-many connection between *Bin* and the set of *Things* (14). It is sufficient to only define a lower bound for this connection and in conjunction with the binary connections we can now compute an upper bound of 40 for a *Bin*, which would occur in a situation where every *Thing* would be put in a separate *Bin*.

5   Conclusion and Future Work

We presented the core of LoCo, a high-level language for modelling configuration problems, including the conditional generation of components. The key feature of the formalism is that the number of components used in configurations is bounded implicitly by the possible number of connections between components. As a next step we plan to extend LoCo so that it is possible to:

- express that the presence of one connection in a configuration depends on the presence of some other connection;

- specify arbitrary combinations of components in the rules for one-to-many connections; and

- incorporate a component taxonomy, where components can be subkinds of other components.

Once this is completed we plan to translate LoCo to an executable format such as SAT, OPL or answer set solving. We also intend to carefully analyse the complexity of e.g. model construction or the bounds propagation algorithm in LoCo.

References

[1] Martin Buchheit, Rüdiger Klein & Werner Nutt (1995): *Constructive Problem Solving: A Model Construction Approach towards Configuration*. Technical Report TM-95-01, DFKI. Available at `ftp://ftp.dfki.uni-kl.de/pub/Publications/TechnicalMemos/1995/TM-95-01.ps.gz`.

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

[3] Andreas Falkner, Ingo Feinerer, Gernot Salzer & Gottfried Schenner (2010): *Computing Product Configurations via UML and Integer Linear Programming*. *Journal of Mass Customisation* 3(4), pp. 351–367, doi:10.1504/IJMASSC.2010.037650.

[4] Gerhard Friedrich & Markus Stumptner (1999): *Consistency-Based Configuration*. In: *Configuration Workshop at AAAI'99*.

[5] Georg Gottlob, Gianluigi Greco & Toni Mancini (2007): *Conditional Constraint Satisfaction: Logical Foundations and Complexity*. In: *IJCAI'07*.

[6] Ulrich Junker (2006): *Configuration*. In F. Rossi, P. van Beek & T. Walsh, editors: *Handbook of Constraint Programming*, Elsevier, pp. 837 – 874.

[7] Phokion G. Kolaitis & Moshe Y. Vardi (1998): *Conjunctive-Query Containment and Constraint Satisfaction*. In: *PODS'98*, doi:10.1145/275487.275511.

[8] Deborah L. McGuinness & Jon R. Wright (1998): *Conceptual modelling for configuration: A description logic-based approach*. *AI EDAM* 12(4), pp. 333–344, doi:10.1017/S089006049812406X.

[9] S. Mittal & B. Falkenhainer (1990): *Dynamic constraint satisfaction problems*. In: *AAAI'90*.

[10] Sanjay Mittal & Felix Frayman (1989): *Towards a Generic Model of Configuration Tasks*. In: *IJCAI'89*.

[11] Daniel Sabin & Eugene C. Freuder (1996): *Configuration as Composite Constraint Satisfaction*. In: *AIMRP'96*.

[12] Daniel Sabin & Rainer Weigel (1998): *Product Configuration Frameworks - A Survey*. *IEEE Intelligent Systems* 13(4), pp. 42–49, doi:10.1109/5254.708432.

[13] M. Stumptner, A. Haselböck & G. Friedrich (1998): *Generative constraint-based configuration of large technical systems*. *AI EDAM* 12(4), pp. 307–320, doi:10.1017/S0890060498124046.

[14] Evgenij Thorstensen (2010): *Capturing configuration*. In: *Doctoral Program at CP'10*.