The Logic of ILOG (J)Configurator: Combining Constraint Programming with a Description Logic

Ulrich Junker
ILOG S.A.
1681, route des Dolines
06560 Valbonne
France
ujunker@ilog.fr

Daniel Mailharro
ILOG S.A.
9, rue de Verdun
BP 85
F-94253 Gentilly Cedex
dmailharro@ilog.fr

Abstract

This paper describes the logic of ILOG (J)Configurator, a powerful configuration system combining constraint programming with a description logic. This combination is based on two key ideas: 1. Constraints are embedded into a logical language and can thus be applied to logical terms formed with the vocabulary of the description logic. 2. When solving the problem, the constructs of description logic are translated into constraints that do the classification while interacting with the other configuration constraints in a tight way.

Keywords: configuration, description logic, constraint programming

1 Introduction

Configuration has become a popular application field for advanced problem solving formalisms such as description logic and constraint programming. Description logic is well-suited to describe the taxonomic knowledge about the system that should be configured. It is able to classify this partially specified system by using knowledge about the classes (e.g. value restrictions) and to complete it by parts in order to meet restrictions about minimum cardinality. Constraint programming is well-suited to treat complex configuration constraints such as compatibility- and requirement-constraints in addition to standard arithmetic, comparison-, and boolean constraints. Local consistency techniques such as arc-consistency are well-adapted for interactive application since they reduce the domains from which users pick their choices.

Most configuration problems involve taxonomic knowledge as well as complex constraints. As consequence, neither description logic, nor constraint programming alone are sufficient to treat those problems. Description logic is often used together with a rule-based system to increase reasoning power, whereas constraint programming is extended in order to be able to treat hierarchical knowledge and to dynamically generate variables for the initially unknown parts of the system. According to our knowledge, there have not been many attempts to combine a description logic with constraint programming. This is quite astonishing since such a combination preserves the advantages of each formalism, while avoiding special treatments to handle the disadvantages. In this paper, we argue that a combined approach has advantages for both communities:

- The description logic community could profit from the high expressiveness and the wealth problem solving algorithms that are elaborated in constraint programming.
- The constraint programming community could profit from well-elaborated methods for classifying classes and for generating objects as provided by DL.

We demonstrate these advantages by giving a concrete example, namely the logic that is employed by ILOG (J)Configurator, a powerful industrial configuration system. (J)Configurator combines a rich constraint language with a description logic similar to $\mathcal{FPC}$ [Magro *et al.*, 2002]. Although the expressiveness of this description logic is quite limited, it is well-suited to describe taxonomic and partonomic hierarchies of configuration problems. Furthermore, we believe that the basic principles of the combinations also applies to more expressive description logics (cf. [McGuiness and Wright, 1998; Horrocks *et al.*, 2002]).

In order to achieve the combination, we proceed as simply as possible by combining the logical language of description logic with a logical language for constraint programming. For example, a characterization of description logic with concepts of predicate logic has been given in [Borgida, 1996]. A logical view of constraint programming has, for example, been elaborated by [Mackworth, 1992]. According to it, each constraint class in the constraint language or constraint library can be considered an n-ary predicate that has fixed interpretation (semantics). For example, the constraint `member(t,T)` between two terms is interpreted by a binary relation, namely the set membership. Furthermore, we model the roles or slots in the description logic by unary function symbols. For example, the tuner and the speakers of a Hifi-system `myAudio` are denoted by `tuner(myAudio)` or `speakers(myAudio)`. The mixed language thus takes its predicates from the constraint language and its function symbols from the description logic. As result, we can formulate constraints on the terms that can be formed with the vocabulary of the description logic. In fact, all positive literals of our language have the form $\sigma(\tau_1, \ldots, \tau_n)$ where σ is a predicate from the constraint library and the τ_i's are terms formed with the function symbols of the description logic as well as logical variables. For example, consider following constraint about the price of the Hifi-system: `price(myAudio) = price(tuner(myAudio)) + sum(speakers(myAudio), price)`. We consider constraints that are either positive literals or negation, conjunction, disjunction, and universal quantification of other constraints. Since the predicates have a fixed interpretation, we only need to interpret the ground (i.e. variable-free) terms τ_i when determining a logical interpretation. Description logic also defines the domains for interpreting these parts. The discussion above can also be applied to restrictions between classes (e.g. subsumption relations, value restrictions) that can be considered constraints on classes. For example, since a class C_i is interpreted by a set S_i of objects, a subsumption relation $C_1 \sqsubseteq C_2$ can be considered a constraint that is interpreted by the subset-relation between S_1 and S_2: $S_1 \subseteq S_2$

Thus, logic shows how to get a combined syntax for description logic and constraint programming. In order to solve a configuration problem formulated in this combined language, we need to find an interpretation of the relevant terms τ_i. This means we have to choose a value for them from their domain such that the constraints are respected. We thus naturally obtain a constraint satisfaction problem where the ground terms τ_i play the role of variables (which must not be confused with the logical variables). For example, the ground term `speakers(myAmplifier)` is mapped to a constraint variable since the speakers of instance `myAmplifier` are chosen by the solving algorithm. We can also treat the constraints from description logic in this way (e.g. subsumption relation, value restrictions) by introducing additional constrained variables for the set of instances of a class and the type of an instance as described in [Mailharro, 1998]. When solving the problem, the constructs of description logic are thus mapped to concepts from constraint programming. This permits a close interaction between inheritance, classification, and propagation of other constraint. Furthermore, description logic can profit from the wealth of constraint solving algorithms including arc consistency, backtracking, non-chronological search, local search, decomposition methods, and so on. This paper thus shows that complex configuration tools can have a natural and simple logical semantics while providing powerful problem-solving algorithms.

The paper is organized as follows. Section 2 introduces the description logic followed by a summary of the constraint language in Section 3. Section 4 discusses different reasoning tasks for configuration problems formulated in the mixed CP/DL-language. Section 5 shows how the constructs from description logic are mapped to constraint programming for solving the problem. We suppose some familiarity with using predicate logic for knowledge representation as it is exposed in standard textbooks (e.g. [Genesereth and Nilsson, 1987]). To illustrate the different concepts in following sections, we will use a simple audio system configuration example, where the problem consists in assembling an amplifier with a set of audio items such as speakers, tuners, CD-players, and to select for each of them a model (a concrete type in fact) from a given catalog. These audio items are submitted to constraints on their price, their quality level, their power, and on compatibility or requirement between them.

2 The Description Logic of Configurator

In this section, we introduce the description logic of (J)Configurator. It is a hierarchical frame-based logic similar to $\mathcal{FPC}$ [Magro *et al.*, 2002]. Although its expressiveness is quite limited, it is well-suited to model the taxonomic and partonomic knowledge found in configuration.

First, we discuss the class hierarchies of (J)Configurator, which consists of concrete classes and abstract classes. Abstract classes have subclasses, whereas concrete classes don't have subclasses. The concrete classes are mutually disjoint and correspond to the concrete part types listed in product catalogs. Two abstract classes are either mutually disjoint or one of the classes is a subclass of the others. Classes have necessary conditions (i.e. conditions that must be satisfied by all instances of the class), but cannot have sufficient ones. Given an object, we thus may detect that it cannot be an instance of a certain class, but we cannot detect that it must be an instance of a class. In our example, the type of the amplifier must be selected among the available models found in a catalog SY2478, AK440, BO93100, MZ9914, that have respective power value: 70, 70, 100, 140. A necessary condition of the class SY2478 is that each instance of it has a power equal to 70, but it is not a sufficient condition since an amplifier that has a power of 70 can be classified either as an SY2478 or as an AK440.

Hence, concrete classes are not defined as specializations of abstract classes, but abstract classes are obtained by aggregating concrete classes. This observation lead us to a description logic consisting of following kind of classes and class expressions:

1. A concrete class is a primitive class. The primitive classes are specified in form of a set $\mathcal{P}$ that satisfied following property: the elements of $\mathcal{P}$ are mutually disjoint ($C_1 \sqsubseteq \neg C_2$).
2. An abstract class is a union $C_1 \sqcup \ldots \sqcup C_k$ of child classes, which are primitive or other abstract classes. The abstract classes are specified in form of a set $\mathcal{A}$ that has two properties:
   (a) if $C_1 \sqcup \ldots \sqcup C_k$ is in $\mathcal{A}$ then all C_i's are either in $\mathcal{A}$ or in $\mathcal{P}$.
   (b) $\mathcal{A}$ is hierarchical meaning that each C can appear in at most one class union that is an element of $\mathcal{A}$.

These are the only class expressions, for which we create instances. In particular, the design of (J)Configurator does not permit the creation of new classes that are intersections or the complement of other classes. We can, however, use other class expressions to formulate necessary conditions on the classes in $\mathcal{A}$ and $\mathcal{P}$ as we will show below.

Next, we consider properties of class instances, which model attributes of components and relations between components. A property P is described by a binary predicate symbol which relates an object to other objects or elements of a data-type domain. A property has a definition class and relates only instances of this definition class to other objects. For example, we can have an object property `hasSpeaker` describing a relation between two objects (e.g. `hasSpeaker(myAmplifier, speaker1)`) or a datatype property `hasPrice` between an object and an integer (e.g. `hasPrice(speaker1, 500)`). We further distinguish between single-valued and multi-valued properties. For example, `hasPrice` is a single-valued property since it relates an audio item with a single price (we cannot have `hasPrice(speaker1, 500)` and `hasPrice(speaker1, 800)`. However, the property `hasSpeaker` is a multi-valued property since an amplifier can be connected to several speakers.

In addition to these given properties, we introduce a property called *type* representing the type of an object, i.e. the concrete class to which it belongs. The type of an object o is interpreted by one of the concrete classes in $\mathcal{P}$, namely the class C that has o as instance.

In order to simplify chaining of properties, we replace the binary predicate symbols by unary function symbols. Whereas the predicate symbols are interpreted by binary relations, the function symbols are interpreted by functions mapping objects to values or set of values. Throughout this paper, we denote the interpretation of a language construct e by $I(e)$, thus following standard conventions for the semantics of predicate logic (cf. [Genesereth and Nilsson, 1987]):

1. A single-valued property P is transformed into a single-valued function π_P. A literal of the form $P(x, y)$ is transformed into $\pi_P(x) = y$ where $=$ is a special equality-predicate. The semantics of π_P depends on the semantics of P:

$$\begin{array}{c} I(\pi_P)(u) = v \\ \text{iff} \\ \text{for all}(u', v') \in I(P) : \text{if } u' = u \text{ then } v' = v \end{array} \tag{1}$$

2. A multi-valued property P is transformed into a multi-valued function Π_P. A literal of the form $P(x, y)$ is transformed into $y \in \Pi_P(x)$ where $\in$ is a special set-membership-predicate. The semantics of Π_P depends on the semantics of P:

$$I(\Pi_P)(u) = \{v \mid (u, v) \in I(P)\} \tag{2}$$

The functions on objects are called fields in (J)Configurator. The domain of the function determines the kind of the field:

1. Single-valued fields:
   - numeric fields have an integer, decimal, or floating-point domain which is specified by an interval or an enumeration of values. `price(tuner)` or `power(amplifier)` are examples or numeric fields[1].
   - string fields have a string domain which is specified by an enumeration of values. An example could be `quality(amplifier)` which value must be selected among “high”, “medium”, “low”.
   - object fields have an object domain, namely the set of instances of a given target class. They typically represent unary relations between objects such as `tuner(amplifier)`.
   - class field have a class domain, namely the set of concrete classes that are subclasses of a given target class. An example is the classification field of the objects: `type(amplifier)`. In all other cases, a class field represents an anonymous instance of a class.
2. Multi-valued fields:
   - object-set fields map an object to a set of objects in a given object domain. They typically represent n-ary relations between objects such as `speakers(amplifier)`.
   - class-set fields map an object to a set of classes in a given class domain. An example could be the set of options selected for an amplifier `options(amplifier)`. A class-set field represents a set of anonymous instances of a class.
   - class-bag fields map an object to a bag of classes (i.e. a multi-set). Class bags are well-suited to represent a selection of part types from product catalogs and their quantities.

Multi-valued fields have an associated cardinality `cardinality`.

(J)Configurator provides the possibility to formulate cardinality and domain restrictions on classes. These restrictions have the form $C \sqsubseteq E$ where C is a class and E is a class expression of following kind:

- toClass $\forall P.C'$ meaning that property P relates the instances of C only with instances of C'. An example, could be that for the SY2478 amplifier, the set of possible speaker types is reduced to the models of a given constructor.
- toDomain $\forall P.D$ meaning that property P relates the instances of C only with elements of domain D. An example, is the reduction of power value for each type of amplifier.
- minCardinality $\geq nP.C'$. For example, the amplifier MZ9914 requires at least 2 speakers of the constructor MZ.
- maxCardinality $\leq nP.C'$. For example, the amplifier MZ9914 cannot support more than 2 speakers of the constructor BO.

These restrictions thus correspond to necessary conditions on the classes.

Configuration does not only require the representation of a taxonomic hierarchy, but also of a partonomic hierarchy. A part-whole relation between instances of a class A and instances of a class B can be represented by a property. A partonomic hierarchy consists of several such properties. We require that this partonomic hierarchy is acyclic: If an instance of A contains an instance of class B then an instance of class B cannot contain an instance of class A. We formulate this as follows. Let $\succ$ be a strict partial order on the classes. A set of

[1]Here, $\Pi_{hasPrice}$ corresponds to the function symbol `price`. Similar correspondences hold for the other examples.

properties $\mathcal{H}$ is called a *partonomic hierarchy* if all properties P in $\mathcal{H}$ relate instances of a class C_1 to a class C_2 such that $C_1 \succ C_2$.

We can refer to parts of single-values properties by the property term $\pi_P(o)$. These terms can also be nested and thus allow us to address indirect parts. Similarly, we can refer to the set of parts defined by multi-valued properties by the property term $\Pi_P(o)$. Again these terms can be nested, but do not allow to refer to an individual term. Without restriction of generality, we can assume that the elements in $\Pi_P(o)$ are totally ordered. We now introduce additional functions for addressing the i-th element in $\Pi_P(o)$, namely $\pi_{p,i}(o)$. For example, the first speaker of the AudioSystem `a` is called `speaker1(a)` and the second speaker is called `speaker2(a)`. Thus, a partonomic hierarchy $\mathcal{H}$ defines a set $\mathcal{F}_\mathcal{H}$ of function symbols that permits to refer to parts. The set $\mathcal{F}_\mathcal{H}$ contains following elements:

1. π_P for each single-valued property P in $\mathcal{H}$.
2. $\pi_{p,i}$ for each multi-valued property P in $\mathcal{H}$ and each $i = 1, \ldots, k$ where k is the maximal cardinality for property P.

Finally, we discuss how the object domains are defined. We distinguish several policies to define this universe.

1. **Explicit Universe:** this policy requires that the user explicitly defines a set $\mathcal{O}$ of object (symbols). These objects will be interpreted by themselves meaning that two different object symbols name different objects (Unique names assumption). No other objects than in $\mathcal{O}$ will be considered in this case. Hence, it is necessary to include root objects (of a partonomic hierarchy) as well as their parts in $\mathcal{O}$. This universe is finite. Using this policy for the audio system configuration problem would consist in generating a priori a maximal set of audio items, among which a subset must be selected to define a solution of the problem. The difficulty is to define a sufficiently large set of objects.
2. **Partonomic Herbrand Universe:** the second policy is based on a partonomic hierarchy $\mathcal{H}$ and a set of root objects $\mathcal{O}$ where each root object r belongs to an initially chosen class r_C. We consider the set of terms that can be formed with the root objects and the function symbols $\mathcal{F}_\mathcal{H}$ that name parts of the partonomic hierarchy ways. Let the set $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ be the smallest set satisfying following conditions:
   (a) each root object of $\mathcal{O}$ is in $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ and is an instance of r_C.
   (b) if P is a single-valued partonomic relation in $\mathcal{H}$ with definition class C_1 and target class C_2 and the term τ from $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ is a possible instance of C_1 then the term $\pi_P(\tau)$ is in $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ and is a possible instance of C_2.
   (c) if P is a multi-valued partonomic relation with definition class C_1, target class C_2 and maximal cardinality k and τ from $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ is a possible instance of C_1 then $\pi_{P,i}(\tau)$ is in $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ for $i = 1, \ldots, k$ and is a possible instance of C_2.

Syntax	Semantics
`negate(`μ`)`	$I(\texttt{negate})(u) = -u$
`sum(`μ,ν`)`	$I(\texttt{sum})(u, v) = u + v$
`difference(`μ,ν`)`	$I(\texttt{difference})(u, v) = u - v$
`product(`μ,ν`)`	$I(\texttt{product})(u, v) = u * v$
`quotient(`μ,ν`)`	$I(\texttt{quotient})(u, v) = u/v$
`abs(`μ`)`	$I(\texttt{abs})(v) = \mathit{abs}(v)$
`min(`μ,ν`)`	$I(\texttt{min})(u, v) = \mathit{min}(u, v)$
`max(`μ,ν`)`	$I(\texttt{max})(u, v) = \mathit{max}(u, v)$
`cardinality(`Ω`)`	$I(\texttt{cardinality})(O) = \mid O \mid$

Table 1: Numeric expressions.

   Hence, a function symbol π_P or $\pi_{P,i}$ will only be applied to a term if this term can be specialized to an instance of the definition class of P.

   The terms in $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ will be interpreted by themselves and we choose $\mathcal{T}_{\mathcal{O},\mathcal{H}}$ as the universe. We can also say that this universe is the Herbrand universe applied to the root objects and function symbols that correspond to partonomic properties. This universe is finite given an acyclic partonomic hierarchy and appropriate cardinality restrictions. If we apply this policy to the audio system configuration problem, initially, we only generate the root object, that is, the audio system `mySystem`, and then let the configurator manipulate and generate the sub-parts `amplifier(mySystem)`, `tuner(amplifier(mySystem))` or `speaker1(amplifier(mySystem))` in the limits of the universe definition $\mathcal{T}_{\mathcal{O},\mathcal{H}}$.
3. **Implicit Universe:** in this case, the object domain can be freely chosen and the resulting universe can be infinite.

There is a strict hierarchies between these three notions: there are configuration problem that are solved by a partonomic Herbrand universe, but not by an explicit universe since the latter universe may be too small. Similarly, there can be configuration problems that can be solved by an implicit universe, but not with a partonomic Herbrand universe. In the sequel, we limit our discussion to the partonomic Herbrand universe.

3 Constraint Programming

In this section, we introduce the constraint language and show how it can be combined with the description logic presented in the last section.

(J)Configurator incorporates a rich constraint language including numeric as well as symbolic constraints. The language consists of predicate and function symbols having a fixed signature and semantics. The signature defines the number and types of the arguments, which correspond to the different types of domains. For the function symbols, the signature also defines the result type. The same function symbol can have several signatures. For example, (sum) can be used for the sum of two integer expressions, but also for a sum-over-set expression. The signatures are different in both cases.

Syntax	Semantics
`union(`Ω_1`,`Ω_2`)`	$I(\texttt{union})(O_1, O_2) = O_1 \cup O_2$
`intersect(`Ω_1`,`Ω_2`)`	$I(\texttt{intersect})(O_1, O_2) = O_1 \cap O_2$
`difference(`Ω_1`,`Ω_2`)`	$I(\texttt{difference})(O_1, O_2) = O_1 - O_2$
`subset(`Ω_1`,`Ω_2`)`	$I(\texttt{subset})(O_1, O_2) = 1$ iff $O_1 \subset O_2$
`subseteq(`Ω_1`,`Ω_2`)`	$I(\texttt{subseteq})(O_1, O_2) = 1$ iff $O_1 \subseteq O_2$
`member(`ω_1`,`Ω_2`)`	$I(\texttt{member})(o_1, O_2) = 1$ iff $o_1 \in O_2$

Table 2: Set-expressions and -constraints.

Syntax	Semantics
`sum(`Ω`, `π_P`)`	$I(\texttt{sum})(O, f) = \sum_{i \in O} f(i)$
`min(`Ω`, `π_P`)`	$I(\texttt{min})(O, f) = \mathit{min}\{f(i) \mid i \in O\}$
`max(`Ω`, `π_P`)`	$I(\texttt{max})(O, f) = \mathit{max}\{f(i) \mid i \in O\}$
`union(`Ω`, `Π_P`)`	$I(\texttt{union})(O, F) = \bigcup_{i \in O} F(i)$
`setof(`Ω`, `C`)`	$I(\texttt{setof})(O, I_C) = \{i \in O \mid i \in I_C\}$

Table 3: Expressions over sets

In contrast to standard approaches to constraint satisfaction, these predicate and function symbols are not applied to existentially quantified variables such as

$$\exists x_1, \ldots, x_n : \bigwedge_i c_i(x_1, \ldots, x_n) \tag{3}$$

but onto the property-terms that can be formed with the vocabulary of the description logic as described in the last section. We have seen that these terms have an associated type and that we can distinguish between numeric, string, object-, class-, object-set-, class-set- and class-bag-terms.

Given the property-terms, we can now apply the function symbols of the constraint language in order to formulate new terms, which we call expressions. Let f be an n-ary function symbol of the constraint language. This function symbol is used to construct expressions of the form $f(\tau_1, \ldots, \tau_n)$ and is interpreted by a function $I(f)$. We distinguish different kinds of expressions depending on the argument and result type.

1. Numeric expressions can be formulated with the binary functions symbols `sum`, `difference`, `product`, `division` and the unary function `negate` that are applied to other numeric expressions and that have obvious interpretations in form of arithmetic operators. Table 1 gives the signature of the function symbols and the semantics. We use μ, ν for numerical expressions and u, v for the numerical arguments[2] of the function $I(f)$.

2. Object- and class-set expressions can be formulated with the binary function symbols `union`, `intersection`, `difference` that are applied to other set expression. Table 2 shows syntax and semantics. We use Ω for object-set expressions and O for object sets. Similarly, we use ω for object expressions and o for objects. C denotes a class and I_C denotes a set of objects interpreting this class.

3. The unary function symbol `cardinality` maps an object- or class-set expression to an integer expression, namely its cardinality.

In addition to these expressions, there are expressions that directly refer to the properties of the description logic. For example, a sum-over-set sums up the value of a numeric property of all objects in an object-set expression. A setof-constraint determines the subset of objects that have the type c. Table 3 summarizes these constraints and their semantics. In addition to the previous convention, f stands for a function mapping an object to a numerical property and F represents a function mapping an object to a set of objects. There is also a setof-constraint that determines the set of objects satisfying a given constraint and a cardof-constraint that determines the number of those objects. The interpretation of this constructs is very subtle and has been omitted for the sake of brevity.

The function symbols thus enable us to form new terms out of the property-terms considered in the last section. We introduced property terms for attributes of objects such as the power of an amplifier `power(myAmplifier)`, for relation between objects such as `tuner(myAmplifier)` or `speakers(myAmplifier)`, and for the the classification of an object `type(myAmplifier)`. A term can also represent an expression on these property-terms, including, path-expression such as `price(tuner(myAmplifier))`, aggregation expression such as `sum(speakers(myAmplifier), price)` or `max(speakers(myAmpli), power)`, arithmetic expression such as `price(myAmplifier) + price(tuner(myAmplifier))`, set expression such as `union(speakers(myAmplifier), cdPlayers(myAmplifier))` or `subset(X in speak-`

[2]It is important to note that the tables specify the interpretation of a function symbol such as `sum` and not the interpretation of the whole term $\texttt{sum}(\mu, \nu)$. For interpreting the term $\texttt{sum}(\mu, \nu)$, we need to first interpret the terms μ, ν and then apply the interpretation of `sum` to them. In this case, u and v will be replaced by the interpretations of μ and ν.

Syntax	Semantics
compatibility(Ω_1 Ω_2,π_P,T)	$I(\texttt{compatibility})(O_1, O_2, f, T) = 1$ iff $\forall i \in O_1, \forall j \in O_2 : (f(i), f(j)) \in T$
requirement(Ω_1,Ω_2,π_P,T)	$I(\texttt{requirement})(O_1, O_2, f, T) = 1$ iff $\forall i \in O_1, \exists j \in O_2 : (f(i), f(j)) \in T$

Table 4: Compatibility and requirement constraints

ers(myAmplifier), (X.price > 100)).

Expressions involving the classification term type(o) compensate the limited expressiveness of our description logic and enable us to formulate more sufficient conditions of classes as well as more complex necessary condition.

As next step, we apply the predicate symbols of the constraint language to the expressions in order to formulate constraints. If σ is an n-ary predicate symbol of our constraint language then the term $\sigma(\tau_1, \ldots, \tau_n)$ is a positive constraint literal. The predicate symbol σ is interpreted by a relation $I(\sigma)$. We distinguish different kinds of constraints depending on the argument type:

1. Numeric constraints can be formulated with the binary predicate symbols eq, ge, le, gt, lt applied on numeric expressions.
2. Set constraints can be formulated with the binary predicate symbols eq, neq, subseteq, member.
3. Symbolic constraints such as the requirement- and compatibility-constraints listed in table 4). Both refer to a table T of tuples.

The final step is to combine the constraint by using standard logical operators and functors. A constraint is either a positive constraint literal or

- the negation of a constraint
- the disjunction or conjunction of constraints
- a forall-constraint of the form forall(x, $\phi(x)$) where x is a variable and $\phi(x)$ is a constraint that may contain free occurrences of this variable.

Following example of an forall-constraint indicates that the power provided by an amplifier needs to be smaller than the minimum of powers required by its speakers:

```
forall X in Ampli:
    power(X) <= min(speakers(X), power);
```

Another example expresses a compatibility constraint between amplifier type and speakers:

```
forall X in Ampli:
    compatibility(type(X),
                  speakers(X),
                  "Ampli-Speaker-Table");
```

We do not consider existential quantifier and existentially quantified constraint. Instead (non-partonomic) properties should be used, which correspond to Skolemized forms of existential quantifier.

We will see later that a kind of unit resolution is supported for the forall-constraint. Since this forall-constraint involves only constraint, we do not need a pattern matching procedure that compares the constituents of the forall-constraint with a data-base of facts. Instead, the unit resolution procedure determines instances that satisfy or violate a sub-constraint.

In the sequel, we consider a set of constraints that does not have free occurrences of logical variables.

	Constants	Function symbols	Predicate Symbols
DL	concrete classes	$\sqcup$	$\sqsubseteq$
objects	root objects	property-functions	$=, \in$
CP	numbers	expression-functions	constraint-predicates

Table 5: Symbols of the combined CP-DL language.

4 Configuration Problems in the Combined Language

In this section, we summarize the symbols of the combined CP-DL language, discuss the semantics of the language, and define several problem solving tasks such as checking the consistency of classes, finding a solution, or explaining a failure. In particular, we will show that a logical approach provides a simple and well-defined characterization of these tasks.

In the previous two sections, we have introduced several sets of function and predicate symbols that are summarized in table 5. We distinguish constants (0-ary function symbols) from other function symbols. The semantics of certain of those symbols is fixed:

1. partonomic terms are interpreted by themselves
2. function and predicate symbols of constraint language have a fixed interpretation

When defining an interpretation, it is therefore sufficient to choose the interpretation of following terms:

1. non-partonomic property-terms (interpreted by objects (sets) or numeric values)
2. classification-terms (interpreted by a concrete class)

The interpretation of concrete classes and abstract classes can be derived from these choices. An interpretation assigns a truth-value to each constraint as defined by the semantics of predicate logic. It is important to note that the truth value of a constraint is entirely determined by the chosen interpretation of the non-partonomic property-terms and the classification-term.

Given a configuration problem formulated in the combined language, we now define several reasoning tasks. Since logic is used as a framework for combining DL and CP, we can

	Basic Terms	Defined Term	Literals
DL	concrete classes	$C_1 \sqcup C_2$	$C \sqsubseteq \forall P.C'$ $C \sqsubseteq \geq nP.C'$ $C \sqsubseteq \leq nP.C'$
objects	root objects	property-term	$\pi_P(o) = v$ $v \in \Pi_P(o)$
CP	property-term	(CP) expression	constraint

Table 6: Constructs of the combined CP-DL language.

use standard logical characterizations of reasoning tasks and avoid to run into pitfalls. Let $\mathcal{C}$ be a set of class restrictions and of generic constraints that do not refer to any root object. Furthermore, let $\mathcal{R}$ be a set of requirements referring to root objects.

- **Consistency of classes:** given a class (expression) C, we consider the problem whether C has an instance w.r.t. $\mathcal{C}$. We introduce a single root object o for this purpose, namely an instance of C and check whether there is an interpretation based on o that satisfies $\mathcal{C}$. It is important to note that this consistency check does not only include class restrictions, but also constraints formulated with predicates of the constraint language. This is even more important since our description logic does not permit the user to formulate sufficient conditions for classes. In order to formulate a condition saying that an object must belong to a class, we need to use constraints.
- **Finding a Configuration:** this problem consists in finding an interpretation that satisfies all the constraints in $\mathcal{C}$ and the requirement in $\mathcal{R}$.
- **Explanation of failure:** if a configuration problem $(\mathcal{C}, \mathcal{R})$ has no solution, the explanation problem consists in finding an inclusion-minimal subset of $\mathcal{R}$ (or $\mathcal{C} \cup \mathcal{R}$) that has no solution.
- **Revision of Configuration:** if a configuration problem $(\mathcal{C}, \mathcal{R})$ has no solution then the revision problem consists in finding an inclusion-maximal subset of $\mathcal{R}$ (or $\mathcal{C} \cup \mathcal{R}$) that has a solution,

Please note that the logical approach facilitates the tasks of explanations and revisions and can deal with situations where different instances of forall-constraints need to be created in different search states when solving the explanation or revision problem. The CP-DL-approach has also advantages over production-rule-approaches to configuration, which show order-dependence and which have no clear notion of a solution, explanation, and revision.

We do not address other tasks such as default reasoning, optimization, finding preferred solutions, which usually need additional knowledge in form of defaults and preferences (cf. [Junker and Mailharro, 2003]).

5 Mapping DL to CP

In this section, we finally discuss how configuration problems formulated in the combined CP-DL-language can essentially be solved. Whereas logic serves as an adequate framework to combine the language, constraint programming provides a powerful algorithmic framework for solving combinatorial problems. Since constraint predicates have a fixed semantics, it is possible to identify mathematical properties of this semantics. Certain of these properties can be exploited as inference rules and implemented in form of efficient algorithms. A representative and didactic example for this exercise is the `allDiff`-constraints that exploits a matching algorithm from graph theory to do inferences [Regin, 1994]. Constraint programming thus associates the constraint predicates with efficient inference algorithms and defines how these algorithms can communicate with each other.

It is a straightforward idea to translate the constructs of DL to constraint programming. This translation is quite simple to achieve. We simply introduce constraint variables for the unknown parts of an interpretation, i.e. the parts that are not fixed initially. According to the last section, this concerns non-partonomic property-terms of the form $\pi_P(\tau)$ and $\Pi_P(\tau)$ and the classification term *type*(τ). For all of them, we introduce a constrained variable:

1. *variables for single-valued properties:* we introduce a variable $x[\pi_P(\tau)]$ with the same domain as the term $\pi_P(\tau)$.
2. *variables for multi-valued properties:* we introduce a set variable $X[\Pi_P(\tau)]$ with the same domain as the term $\Pi_P(\tau)$. Constrained set variables have been introduced in [Puget, 1992] and applied by [Mailharro, 1998] to configuration problems.
3. *classification-terms:* we introduce a constraint variable $x[\mathit{type}(\tau)]$ that has a class domain containing all possible leaf classes of t.

If the property-term t is a partonomic term, the value of this variable can be chosen freely. If the term t is a non-partonomic term it is unknown and the interpretation of the term $\pi_P(\tau)$ is equal to the interpretation of $\pi_P(o)$ if o is the interpretation of t, i.e. $I(\pi_P(\tau)) = I(\pi_P(I(\tau)))$. We introduce an element-constraint $\pi_P(\tau)$ `= element(`t`,` π_P`)` on t and π_P for establishing this relation. Furthermore, we introduce a constraint variable $x[t]$ representing the unknown interpretation $I(\tau)$ of t and the variables $x[\pi_P(o)]$ for each object o that can have the property P. We use similar element-constraints to guarantee $I(\Pi_P(\tau)) = I(\Pi_P(I(\tau)))$ and $I(\mathit{type}(\tau)) = I(\mathit{type}(I(\tau)))$.

Finally, we translate domain restrictions $C \sqsubseteq E$ of classes. They mean that an instance o of class C must satisfy domain- and cardinality-restrictions on a property P expressed by E. We can reformulate these class restrictions by a constraints on $\pi_P(o)$ and $\Pi_P(o)$. For example, a max-cardinality constraint $C \sqsubseteq \leq 3p.C'$ can be expressed as follows: `forall` x `in` C`: cardof(`Π_P`(x)) <= 3`. The class C' does not appear here since we assume that it is the target class of the property P and that all elements in $\Pi_P(o)$ are instances of this class.

We thus described the translation of DL-constructs to constraint programming. For the sake of brevity and clarity, we omit a descriptions of optimizations of the translation such as lazy generation of objects and constraint variables.

In order to solve the resulting constraint programming problem, we interleave (backtrack) search and inference. Following inference methods are applied in each search state:

- Each expression and constraint uses a specialized propagation algorithm that maintains a local consistency (e.g. bound or arc consistency) and that reduces the current variable domains for this purpose.
- For-all-constraints also have a customized propagation algorithm that performs a kind of unit resolution for a constraint of the form `forall` x `in` C: $\neg\phi_1(x) \vee \ldots \vee \neg\phi_n(x)$.

In addition to these techniques, any other method for solving constraint satisfaction problems can be used including local search, non-chronological search, problem decomposition to name a few.

6 Conclusion

In this paper, we described the logic of ILOG (J)Configurator, thus giving a precise characterization of what (J)Configurator is doing. This logic combines constraint programming (CP) with a description logic (DL) and provides an example how such a combination can be obtained:

1. Constraints are embedded into a logical language and formulated on terms that are formed with the vocabulary of the description logic.
2. Constructs of the description logic are translated to CP in order to solve a configuration problem, thus permitting a tight interaction between DL-restrictions and other configuration constraints.

Even if our description logic is very simple, we believe that these principles are interesting for both the CP and DL communities and can be generalized for more complex description logic, which is a topic for future work.

Our discussion also highlights several benefits of a logical view on constraint programming. This view gives us a concise characterization of what an advanced configurator is doing and clarifies the interaction between description logic and CP. It also provides a smooth integration of a forall-constraint and logical operations into CP. Furthermore, logic is able to deal with unknown universes and thus provides a clean and simple semantics to problems such as dynamic object generation that are addressed by dynamic CSPs [Mittal and Falkenhainer, 1990] and generative CSPs [Stumptner *et al.*, 1998]. Finally, the logical view permits us to represent a configuration problem as a set of constraints even in presence of inheritance and object generation, which facilitates certain tasks such as explanation and revision.

There are many issues that we only sketched in this workshop paper and that will further be elaborated in a long version. First of all, we will give a complete description of the syntax and semantics of the combined DL/CP language. Second, we will demonstrate the expressive power of this approach by a non-trivial configuration example. Third, we will further investigate the mapping of DL constructs to CP and address questions about the impact on performance. An interesting question is whether specialized DL algorithms need to be encapsulated inside global constraints in order to provide an efficient classification. Another question is whether this DL-CP mapping can be generalized to a full DL language in order to make CP technology available to the DL community.

References

[Borgida, 1996] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82:353 – 367, 1996.

[Genesereth and Nilsson, 1987] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Los Altos, 1987.

[Horrocks *et al.*, 2002] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. Reviewing the design of DAML+OIL: An ontology language for the semantic web. In *AAAI-02*, pages 792–797. AAAI Press, 2002.

[Junker and Mailharro, 2003] Ulrich Junker and Daniel Mailharro. Preference programming: Advanced problem solving for configuration. *AI-EDAM: Special Issue on Configuration*, 17(1), 2003.

[Mackworth, 1992] A. K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58(1-3):3–20, December 1992.

[Magro *et al.*, 2002] Diego Magro, Pietro Torasso, and Luca Anselma. Problem decomposition in configuration. In *ECAI 2002 Workshop on Configuration*, 2002.

[Mailharro, 1998] Daniel Mailharro. A classification and constraint based framework for configuration. *AI-EDAM: Special Issue on Configuration*, 12(4), 1998.

[McGuiness and Wright, 1998] D. McGuiness and J. Wright. Conceptual modeling for configuration: A description logic-based approach. *AI-EDAM: Special Issue on Configuration*, 12(4), 1998.

[Mittal and Falkenhainer, 1990] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI-90*, pages 25–32, 1990.

[Puget, 1992] J.-F. Puget. PECOS: a high level constraint programming language. In *SPICIS 92*, Singapore, 1992.

[Regin, 1994] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[Stumptner *et al.*, 1998] Markus Stumptner, Gerhard E. Friedrich, and Alois Haselböck. Generative constraint-based configuration of large technical systems. *AI-EDAM (Artificial Intelligence for Engineering, Design, Analysis and Manufacturing)*, 12(4):307–320, 1998.