# Towards Hybrid Techniques for Efficient Declarative Configuration

**Ingo Feinerer**[1]

**Abstract.** During the last decades configuration has been extensively employed in a wide range of application domains, implemented by a multitude of techniques like logics, procedural, object-oriented, or resourced-driven approaches. Especially declarative methods provide the foundation for precise and well-understood semantics for reasoning tasks and allow for a succinct representation of the underlying knowledge base. However, a drawback in using such powerful declarative techniques lies in their computational complexity. In this paper we present a simple declarative framework for configuration in Prolog in order to show the advantages of logic-based techniques but also to identify some challenges for such formalisms. We argue for hybrid systems which combine and utilize efficient techniques from different configuration methodologies under a unified declarative interface.

## 1 INTRODUCTION

In the history of configuration research multiple high-level approaches for the design and solution of configuration tasks have been presented. Declarative systems have had strong proponents due to their expressive semantics based on well investigated and understood logics. Unfortunately the expressive power limits the applicability due to high computational costs, observable for a broad class of implementations [3]. Related topics like reconfiguration unveil further interesting but often computational hard topics [6, 2] for declarative formalisms. In this paper we identify some typical challenges for declarative frameworks and propose a combined hybrid approach which utilizes techniques from other fields, like integer linear programming (ILP), to overcome these. Section 2 presents a simple declarative framework in order to demonstrate the benefits but also the challenges (Section 3) of such a system. Section 4 shows three strategies to deal with the discussed challenges.

## 2 DECLARATIVE FORMULATION

Object-oriented modeling languages provide a natural formalism for the design of configuration systems. Historically entity-relationship diagrams had a strong toehold but during the recent decade class diagrams in the Unified Modeling Language [8] (UML) have seen a substantial growth as a domain-specific language for configuration. [5, 1] Consequently we use UML class diagrams as our starting point and propose a declarative formulation translating its key features into a declarative framework. We chose Prolog for this task as it has an established and well-known semantics and provides several efficient implementations. The main ideas can be easily transfered to other declarative formalisms (e.g. logics or answer-set programming) as well. The following framework was implemented and tested with SWI-Prolog but should work equally well in any other dialect with minor modifications.

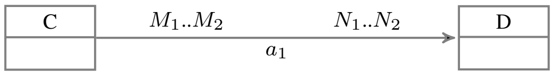


**Figure 1**: Specification with two classes, an association with multiplicities, and the OCL constraint `context C inv: C.allInstances()->size() >= L`.

Figure 1 depicts a minimal UML class diagrams as typically used for configuration purposes. It shows a specification with two classes $C$ and $D$ and one association $a_1$ relating them. The multiplicities restrict the number of valid links; each object of class $C$ must be connected with at least $N_1$ and with at most $N_2$ objects of class $D$. Analogously, each object of class $D$ must have links to $M_1..M_2$ objects of class $C$. The small arrow at the end of the association shows the direction (and not a hierarchy). The constraint in the Object Constraint Language [9] (OCL) for class $C$ states that $C$ must be instantiated with at least $L$ objects to obtain a valid configuration.

The concepts of a class and of an association can be directly translated to

```
class(CN-L) :- atom(CN), L >= 0.

assoc(CNs, AN-(C,I)>>(D,J)) :-
  atom(AN), member(C, CNs), member(D, CNs),
  mult(I), mult(J).
```

where each class has a name (`CN`) and a lower bound (`L`). Each association has a name (`AN`) and includes information on the related classes (`C`, `D`) and the multiplicities (`I`, `J`). Further the class names are checked for validity against a predefined list of names (`CNs`).

Now we can define a specification as a tuple `(Cs, As)` of classes `Cs` and associations `As`

```
spec((Cs, As)) :-
  maplist(class, Cs), pairs_keys(Cs, CNs),
  maplist(assoc(CNs), As).
```

where `maplist()` applies a predicate to all arguments of a list and `pairs_keys()` provides access to the subterms of Pair-Key structures.

E.g., instantiating the multiplicities of $a_1$ in Figure 1 with $M_1 = 1$, $M_2 = 2$, $N_1 = 3$, and $N_2 = 4$ and enforcing a lower bound $L$ of 1 for the number of objects of class $C$ this yields

```
:- spec(([c-1,d-0], [a1-(c,1..2)>>(d,3..4)]))
```

[1] Vienna University of Technology, Austria, email: Ingo.Feinerer@tuwien.ac.at

A configuration can be modeled as a tuple (Os, Ls) consisting of objects Os and of links Ls. Each object is identified by its name O and its corresponding class C. Each link has a name L and stores information on the corresponding association A and on the objects [O,P] it consists of.

```
is_object(CNs, O-C) :-
  atom(O), member(C, CNs).

is_link(As, Os, L-(A,[O,P])) :-
  atom(L), member(A-(C,_)>>(D,_), As),
  member(O-C, Os), member(P-D, Os).
```

We call a configuration an *instance* of a specification if all the objects and links have a corresponding class and association, respectively.

```
instance((Os, Ls), (Cs, As)) :-
  pairs_keys(Cs, CNs),
  maplist(is_object(CNs), Os),
  maplist(is_link(As, Os), Ls).
```

For each object we check its class and for each link we identify a matching association with compatible participating classes.

In order to handle multiplicities as defined by the UML standard, we define a predicate gamma() which counts for each association and a given object the number of different partner objects induced by the links of a configuration [4]

```
gamma(I, P, A, N, Ls) :-
  reduct(I, Os, P),
  findall(Os, member(_-(A,Os), Ls), Bag),
  list_to_set(Bag, Set), length(Set, N).
```

where I denotes the "position" (index) of the partner objects to be counted for within the given binary or multiary association A, the list P contains the single object to be fixed (in general this could be any partial link), and Ls is a list of links to be considered. The result (count) is unified with N. The reduct() predicate generates a list of objects (to be used as partial links) such that P is the projection on all but the I-th component, and findall() collects all such objects forming a link in Ls. For example

```
:- gamma(2, [c1], a1, 3,
         [l1-(a1, [c1, d1]), l2-(a1, [c1, d2]),
          l3-(a1, [c1, d3])]).
```

fixes a single object $c_1$ (of class $C$) and counts how many different objects (of class $D$ corresponding to index 2) can be reached over association $a_1$ when considering the provided links: three.

Now we have all parts to define the notion of a valid configuration. We say a configuration *satisfies* a specification if it is an instance and both the lower bounds for each class and the multiplicities of each association are respected.

```
satisfies((Os, Ls), (Cs, As)) :-
  forall(member(C-LB, Cs),
         (findall(ON, member(ON-C, Os), ONs),
          length(ONs, N), N >= LB)),
  forall((member(O-C, Os),
    (member(AN-(C,_)>>(_,L..U),As),I=2
    ;member(AN-(_,L..U)>>(C,_),As),I=1)),
         (gamma(I, [O], AN, N, Ls),
          between(L, U, N))).
```

The predicate satisfies() checks whether (Os, Ls) is a valid configuration for the specification (Cs, As). The first forall() checks the number of objects for each class against the corresponding lower bound whereas the second forall() checks that each object O of the configuration is linked to the right number of partner objects as constrained by all participating associations and their multiplicities. E.g.

```
:- satisfies(([c1-c, d1-d, d2-d, d3-d],
   [l1-(a1,[c1,d1]),l2-(a1,[c1,d2]),
    l1-(a1,[c1,d3])]),
  ([c-1,d-0], [a1-(c,1..2)>>(d,3..4)])).
```

Although simplistic and minimal this framework allow us to model reasonable configurations and will serve the purpose of presenting the advantages but also challenges for such a declarative framework. It can be easily extended to cover multiple aspects which are necessary for a more complete handling of configurations in real-world applications (and in fact many features are already implemented in a prototype).

A central advantage we observe in the design phase is the clear and straightforward formulation of the underlying terminology. Specifications, configurations, and the notions of instance, validity and satisfiability can be defined with just a few lines of code in parallel to the formal definitions of the underlying concepts. Prototyping is rapid and succinct; the visualization of graph structures and configurations is straightforward. For the computation of configurations one of the main advantages of this declarative formulation is the semi-automatic search for solutions. Prolog provides efficient algorithms to explore and backtrack within the search space; further it has optimizations for tail recursion. E.g., with trivial strategies for object and link creation

```
:- gen_objects([c-2, d-3],
  [c_0-c, c_1-c, d_0-d, d_1-d, d_2-d]).

:- gen_links([a1-1, a2-2],
  [a1_0-(a1,_), a2_0-(a2,_), a2_1-(a2,_)]).
```

we can generate configurations on the fly (we implement the predicate gen_instance() for this purpose) and check if they satisfy a given specification

```
gen_model(C, S) :-
  gen_instance(C, S), satisfies(C, S).
```

For a small number of classes and associations this strategy works very well, is intuitive, and allow us to explore the whole search space without extra programming.

## 3 CHALLENGES

For a declarative framework as presented in the previous section challenges typically arise when the search space is large and/or backtracking is expensive. The former is not a drawback of declarative formulations per se; other formalisms which need to deal with problem instances with a large solution space will suffer from the same performance penalties. However, when backtracking is expensive, i.e., when it takes some computational effort to find out that the current unification state will never lead to a valid configuration, there is often room for improvement. First of all, it depends whether the computational complexity is inherent to the problem or just induced by the use of expressive logics or other declarative formalisms. Second, there exists a plethora of methods and algorithms in computer science which are tailored to specific problem classes.

### 3.1 Equations over association chains

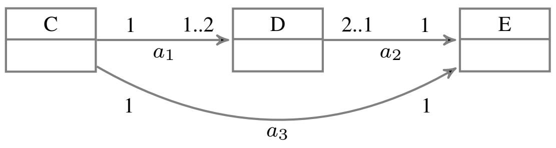
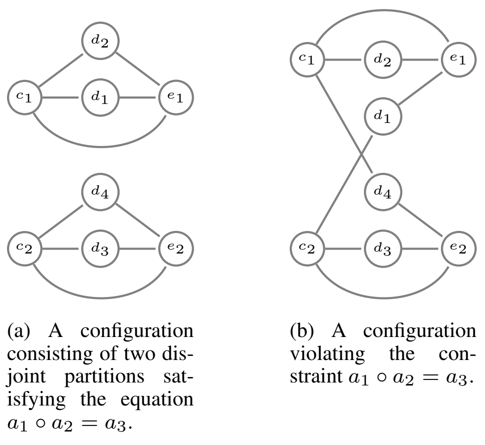


**Figure 2**: Specification with three classes and the OCL constraint `context C inv: C.allInstances()->size() >= 2.`

Consider the specification depicted in Figure 2. There are three classes $C$, $D$, and $E$, where $C$ has a lower bound of two on the number of instantiated objects. Each object of class $C$ should be connected via association $a_1$ to one or two objects of class $D$ which in turn are uniquely associated with an object of class $E$ via $a_2$. Finally association $a_3$ enforces a one-to-one relationship between objects of class $C$ and objects of class $E$.

This poses no severe problem for our declarative implementation yet. We can find a valid configuration by stepwise incrementing the number of instantiated objects and the number of desired links and the `satisfies()` predicate will filter out all invalid combinations. However, the situation gets interesting if we add some constraints; the specification in Figure 2 probably tells not the full story of the intended semantics. We might want to ensure that each object of class $E$ which can be reached from an object of class $C$ over intermediary objects of class $D$ via associations $a_1$ and $a_2$ is in fact the same as linked by $a_3$. This models the concept of a modular compound unit which consists of a set of interconnected subcomponents.

We can formalize such constraints by introducing equations on association chains. They restrict the links of valid configurations by imposing limits on valid objects involved. Association chains can be modeled by classical tuple composition, similar to a join in database theory. E.g., for Figure 2 we would add the equation $a_1 \circ a_2 = a_3$ to achieve the semantics mentioned before.



(a) A configuration consisting of two disjoint partitions satisfying the equation $a_1 \circ a_2 = a_3$.

(b) A configuration violating the constraint $a_1 \circ a_2 = a_3$.

**Figure 3**: Two satisfying configurations for the specification in Figure 2, but only the left one adheres to the constraint $a_1 \circ a_2 = a_3$.

Figures 3a and 3b both show valid configurations for the specification 2 before adding the new constraint on the association chain. Once we enforce the constraint $a_1 \circ a_2 = a_3$ only Figure 3a remains a satisfying instance; Figure 3b is not valid anymore as the configuration violates the equation since there is a connection from $c_1$ over $d_4$ to $e_2$ which is not reachable via a link of association $a_3$.

This subtle change makes a significant difference in runtime and backtrack behavior for our declarative formulation. Although the number of objects and links to be considered is still rather small (8 and 10, respectively) the number of combinations to be explored before the equation can be checked is exponential. Backtracking is therefore very expensive as basically a full configuration needs to be built before the equation can be checked. We see a major increase in the runtime (and stopped measuring after 3600 seconds).

### 3.2 Partial configurations

We would like to use our framework not only for configuration but also for reconfiguration in order to repair existing configurations. A main challenge is a fast way to identify parts of an input configuration which cannot be taken as a subcomponent of the overall solution. This is important to avoid late backtracking where a lot of computation time has been used for building a variety of alternations which can never lead to a valid configuration. Clearly, for arbitrary complex input configurations this problem is NP-hard, however for many basic tasks it is not (e.g., checking whether some links will violate certain sets of constraints later on).

### 3.3 Cost functions

Another limiting factor of logic-based formalisms is finite domain reasoning, especially notable when working with numbers in an integer domain. Configuration tasks often need ways to express how expensive certain components or connections are. A natural implementation is to use cost functions which assign costs or other integer numbers to individual objects in a configuration. The aim is now to minimize an objective function which takes into account all components and their corresponding costs.

## 4 TOWARDS A HYBRID APPROACH

In the previous section we saw some typical challenges for declarative frameworks. These are by far not the only ones but give a sample of relevant problems of various kinds. Fortunately, there has been tremendous progress in configuration research and artificial intelligence in general which triggered specialized algorithms and strategies for specific problems. In order to tackle the outlined challenges we propose to use specialized algorithms from other formalisms and to include them in a unified declarative framework. Such a hybrid approach, taking the best from multiple worlds and combining them in a consolidated interface, allows us to attack the previously described challenges towards efficient declarative configuration.

### 4.1 Integer linear programming

We start out with the observation that backtracking can be very expensive if the domain of variables is not restricted or bounded to a specific range. Ideally we would like to compute the exact number of needed components for a configuration and then our declarative framework can concentrate on finding appropriate links to connect the parts. Clearly, this approach cannot always work as some special constraints on the interconnection of components may enforce additional objects which are not required by pure associations (and their multiplicities) in the underlying UML class diagram.

For the computation of the needed number of objects for each class we use a highly efficient technique based on integer linear programming. [7, 4] The idea is to translate a UML class diagram to a system of inequalities. Its solution indicates whether an instantiation into a valid configuration is possible at all (satisfiability problem) but more interestingly in our context gives also the number of objects for each component. E.g., for the specification in Figure 1 with association $a_1$ and the lower bound on $C$ we obtain

$$\begin{array}{lll} M_1 \cdot |D| \leq N_2 \cdot |C| & |C| \geq M_1 & |C| \geq L \\ N_1 \cdot |C| \leq M_2 \cdot |D| & |D| \geq N_1 & \end{array}$$

expressing constraints enforced by the multiplicities (left), constraints on the minimal number of objects due to the association semantics (middle), and the lower bound constraint on $C$ (right). This translation is performed for all classes and associations involved in a specification forming an integer linear program. The objective function is typically just the minimum over all classes involved.

SWI-Prolog provides support for integer linear programming via its extension library "simplex" shipped with the standard installation. We added a set of DCG (definite clause grammar) rules which generate ILP constraints out of each association:

```
assoc_constraint(
    _AN-(C,M1..M2)>>(D,N1..N2)) -->
  constraint([M1*D, -N2*C] =< 0),
  constraint([N1*C, -M2*D] =< 0).
```

Further constraints like lower bounds or the objective function are added separately forming the whole ILP program.

The native integration of ILP into our declarative framework allows us to rule out a broad range of possibilities which do not need to be explored any more. This has a significant effect on backtracking as it cuts the search space into more fine grained areas. With this technique we can scale our framework to a greater number of objects and links when dealing with the challenges "equations on association chains" and "partial configurations". Costs or weights are an integral part of ILP and can thus easily be handled with this approach; this addresses several aspects of the challenge "cost functions".

## 4.2 Procedural link generation

So far we have only identified one possible way to efficiently compute the number of necessary components. Still, it might take a long time to find valid links between these objects (as motivated in the challenge on association chains). Therefore it seems a promising approach to optimize the way links are generated. One strategy we found useful for certain classes of configurations is to "balance" links between objects as far as possible. This can be seen as a procedural implementation to form a uniform distribution among the links between the participating classes of a given association:

```
seq(J, M, N, [X, Y]) :-
  J >= 0, M > 0, N > 0,
  X is J mod M,
  lcm(M, N, LCM),
  Y is (J + floor(J/LCM)) mod N.
```

The idea of the `seq()` predicate is to generate a sequence (with `J` as the index into it) of tuples `[X,Y]` for `M` objects of the first class and `N` objects of the second class (for a binary association) which can be used as links and form a uniform distribution. E.g., for $J = 0, \dots, 3$ and $M = 2$ and $N = 6$ we obtain the tuples $(0, 0)$, $(1, 1)$, $(0, 2)$, and $(1, 3)$. Consequently we would generate a link from object 0 of class $C$ to object 0 of class $D$, from object 1 of class $C$ to object 1 of class $D$, and so on.

Such a strategy is especially suited for finding an initial linking for a whole configuration. This can be both the basis for further investigation regarding the challenge "equations on association chains" or provide initial solutions for a reconfiguration problem as necessary for the challenge "partial configurations".

## 4.3 Generate and test

Our final suggestion towards hybrid systems is a meta-strategy. Both previous techniques also fall into this category as special cases. Declarative systems have a strong standing with their efficient and native backtracking as it is at the heart of their operation. This property is extremely useful for configuration as exploring the solution space is one of the fundamental aspects in a configuration task. The main advantage is that almost arbitrary constraints can be added with minimal changes to the declarative implementation. This allows us fast prototyping and complex constraint checking. We therefore argue in favor of such systems and the simple but effective strategy of "generate and test". The generation step is either done by more sophisticated algorithms implemented by the user in Prolog (as shown in the previous subsections) or by calls to external tools which are specialized on a specific tasks. The declarative framework can then take these parts and combine them into a configuration and test it for validity.

## 5 CONCLUSION

Declarative formalisms provide a natural environment for the implementation of configuration systems as it is easy to write succinct and precise programs with integrated support to explore the solution space with backtracking. We motivated this by a simple declarative framework but showed challenges arising from their use. We argue for hybrid systems which combine specialized techniques from other fields into a unified declarative interface.

## REFERENCES

[1] Andreas Falkner, Ingo Feinerer, Gernot Salzer, and Gottfried Schenner, 'Computing product configurations via UML and integer linear programming', *Int. Journal of Mass Customisation*, **3**(4), 351–367, (2010).

[2] Andreas Falkner, Gerhard Friedrich, Alois Haselböck, Anna Ryabokon, Gottfried Schenner, and Herwig Schreiner, '(Re)configuration using answer set programming', in *IJCAI 2011 Configuration Workshop*, (2011).

[3] Andreas Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, 'Modeling and solving technical product configuration problems', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **25**, 115–129, (2011).

[4] Ingo Feinerer and Gernot Salzer, 'Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints', in *Proceedings of the 1st IEEE/IFIP International Symposium on Theoretical Aspects of Software Engineering, June 6–8, 2007, Shanghai, China*, pp. 411–420. IEEE Computer Society Press, (2007).

[5] Alexander Felfernig, Gerhard Friedrich, and Dietmar Jannach, 'UML as domain specific language for the construction of knowledge-based configuration systems', *International Journal of Software Engineering and Knowledge Engineering*, **10**(4), 449–469, (2000).

[6] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(Re)configuration based on model generation', in *2nd LoCoCo Workshop*, volume 65 of *EPTCS*, pp. 26–35, (2011).

[7] Maurizio Lenzerini and Paolo Nobili, 'On the satisfiability of dependency constraints in entity-relationship schemata', *Information Systems*, **15**(4), 453–461, (1990).

[8] Object Management Group, *Unified Modeling Language 2.4.1*, 2011.

[9] Object Management Group, *Object Constraint Language 2.3.1*, 2012.