Knowledge Representation Analysis of Graph Mining

Matthias van der Hallen†⋆ , Sergey Paramonov†, Michael Leuschel‡, Gerda Janssens†

†KU Leuven, ‡Heinrich-Heine-Universität Düsseldorf

Abstract. Many problems, especially those with a composite structure, can naturally be expressed in higher order logic. From a KR perspective modeling these problems in an intuitive way is a challenging task. In this paper we study the graph mining problem as an example of a higher order problem. In short, this problem asks us to find a graph that frequently occurs as a subgraph among a set of example graphs. We start from the problem's mathematical definition to solve it in three state-of-the-art specification systems. For IDP and ASP, which have no native support for higher order logic, we propose the use of encoding techniques such as the disjoint union technique and the saturation technique. ProB benefits from the higher order support for sets. We compare the performance of the three approaches to get an idea of the overhead of the higher order support.
We propose higher-order language extensions for IDP-like specification languages and discuss what kind of solver support is needed. Native higher order shifts the burden of rewriting specifications using encoding techniques from the user to the solver itself.

1 Introduction

Many real world problems exhibit a composite structure consisting of multiple smaller problems which can be combined in many different configurations. These types of problems lend themselves for a declarative approach as knowledge representation offers a transparent, natural and extendable model satisfying 'The Principle of Elaboration Tolerance' [McCarthy, 1998]: declarative specifications are easily adapted to new requirements or changed circumstances, e.g. variations in which subproblems are used, and in the way they are combined. Conversely, the smaller problems in these composite structures are often already NP or coNP complete. Combining these already complex problems often raises the computational complexity of the composite problem, up to a level where it cannot be expressed using first order logic. These problems become higher order logic problems: We study the *Graph Mining* problem as an example featuring such a raise in complexity.

Specification languages with support for higher order logic exist, with different levels of support. On the one hand, meta-programming, as known from Logic Programming [Abramson and Rogers, 1989], has inspired the introduction of higher-order atoms in DLVHex [Eiter et al., 2005] and the higher-order syntax in HiLog [Chen et al., 1993]. As in Prolog, predicate symbols can be either constants (first order case) or variables (second order case). In the case of predicate variable symbols, these variables range over predicate names, and not the predicate space itself, essentially combining second order syntax with first order semantics. This cannot model the graph mining problem.

⋆ Matthias van der Hallen is supported by a Ph.D. fellowship from the Research Foundation - Flanders (FWO - Vlaanderen).

On the other hand, formal specification languages such as Z [Bowen, 1996], B [Abrial, 1996], Event-B [Abrial, 2010] and TLA [Lamport, 2002] extend predicate logic with set theory and offer higher order datastructures. ProB [Leuschel and Butler, 2008] is a constraint solver, animator and model checker for such languages, implemented in SICStus Prolog. We can express the graph mining problem in ProB directly using higher order logic, but in general such systems miss the flexibility to perform multiple different inferences such as model expansion and optimization without modifying the specification. Furthermore, ProB requires an encoding to express inductive definitions, and as it is built on CP techniques and finite domain solvers, it does not benefit from the recent revolutions in solving techniques such as CDCL.

Therefore, we also look at specification languages that do not allow higher order syntax. Examples of such languages are the IDP [De Cat et al., 2016] and the ASP [Eiter et al., 2009] language. For these languages, several techniques exist that allow the user to simulate higher order logic to model problems such as graph mining, potentially offering better performance than systems that allow higher order logic directly.

Graph mining is a specific kind of *frequent pattern mining*, the task of enumerating patterns which occur frequently in a dataset. A first class of *pattern mining* is *unstructured mining*, such as *itemset mining*, where the pattern is a set of items without any additional structural relation between the different items. This problem is of propositional nature: De Raedt et al. [2008] modeled it using CP techniques, while Järvisalo [2011] used ASP. Recently, focus has shifted from unstructured towards structured mining, such as graph or sequence mining Négrevergne and Guns [2015], Gebser et al. [2016]. Here, the items being mined exhibit additional structure, for example the edge relation in the case of graph mining. This introduces the$\mathbf{NP}$-coplete problem of graph homomorphism [Levin, 1973], and its many variations, which in imperative languages lead to many different algorithms [Yan and Han, 2002, Dries and Nijssen, 2012]. A declarative approach can express these variations with only minimal changes.

In our case study of the graph mining problem, we start with from the mathematical model of graph mining, which is inherently higher order, and identify the following contributions:

- We identify the higher order aspects of the graph mining problem and show how the problem can be modeled in IDP, ASP and ProB, proposing concrete modeling techniques. We also identify a set of desirable properties for a declarative encoding of the graph mining problem.
- We propose a higher order encoding that closely follows the mathematical model of graph mining, and satisfies all desirable properties of a declarative graph mining model. We indicate how additional solver support can exploit the additional structure in this encoding to work more efficiently.

The paper is structured as follows: Section 2 introduces graph mining formally, Section 3 discusses the how to model the problem in IDP, ASP and ProB, identifying a set of desirable properties. Then, Section 4 discusses the performance of these systems. Section 5 discusses a faithful encoding of the graph mining problem in an KR language enriched with HO, and its possible solver implementation. Section 6 draws conclusions and outlines possible future research directions.

2 Formalization of the graph mining problem

2.1 Patterns

We start with a comprehensive formal definition of the graph mining problem.

Definition 1. *A labeled graph $\mathcal{G}$ is a triple $\langle V, E, l\rangle$ where V is the finite set of vertices or nodes, E is a binary predicate on V that represents the set of (directed) edges and l is a unary function from V to a set of labels.*

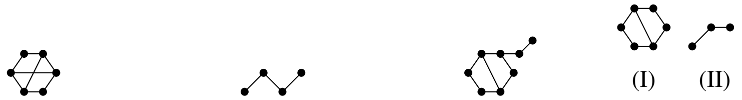
Definition 2. *A graph $\mathcal{G} = \langle V, E, l\rangle$ is* connected *iff for each pair of vertices v and v' in V, there exists an edge $(v, v') \in E$ or there exists a sequence $v, v_1 \ldots v_n, v'$ such that there exist edges (v, v_1), (v_i, v_{i+1}) and $(v_n, v') \in E$, where $1 \leq i \leq n - 1$.*

Definition 3. *A graph homomorphism f from a labeled graph $\mathcal{G} = (V, E, l)$ to a labeled graph $\mathcal{G}' = (V', E', l')$ is an injective mapping $f : V \rightarrow V'$ from vertices of $\mathcal{G}$ to vertices of $\mathcal{G}'$ such that:*
- $\forall v \in V : l(v) = l(f(v))$ *(the mapping respects labelings), and*
- $\forall u, v \in V, (u, v) \in E \implies (f(u), f(v)) \in E'$ *(the mapping preserves edges).*

If a graph homomorphism from graph $\mathcal{G}$ to $\mathcal{G}'$ exists we say $\mathcal{G}$ is homomorphic *with $\mathcal{G}'$.*

Definition 4. *Given a pair $\langle \mathbb{E}_+, \mathbb{E}_-\rangle$ consisting of a set of* positive *and* negative *examples of* labeled graphs *respectively, and a graph $\mathcal{T}$ called the template,* Graph mining *is the problem of finding a pattern $\mathcal{P}$ which is*
- *a* connected labeled subgraph of *$\mathcal{T}$,*
- homomorphic *with at least N_+ positive examples $\mathcal{E}_+ \in \mathbb{E}_+$, while being homomorphic with at most N_- negative examples $\mathcal{E}_- \in \mathbb{E}_-$.*

We call these homomorphisms the positive (negative) homomorphisms, and the restriction on their number the positive (negative) homomorphic property, respectively.



(a) Positive Example (b) Negative Example (c) Template (d) Pattern Candidates

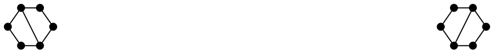
Fig. 1: A graph mining instance ($N_+ = 1, N_- = 0$) with pattern candidates.

Take, for example, the problem set shown in **Fig.** 1. We assume all nodes have the same label, and that all edges are bidirectional. The template graph guides the search. There is one positive example (**Fig.** 1a), and one negative example (**Fig.** 1b). **Fig.** 1c shows the template graph. **Fig.** 1d shows a valid and an invalid pattern. They are both connected subgraphs of the template. Requiring at least one homomorphism with a positive example, and allowing no homomorphisms with negative examples (i.e. problem parameters $N_+ = 1$ and $N_- = 0$), **Fig.** 1I represents a valid pattern. It is clear that there exists a mapping from each node from the valid pattern to a node of the positive example, while no such mapping exists for the negative example. Looking at **Fig.** 1II, this graph is clearly homomorphic with both the positive as well as the negative example. Therefore, it is not a pattern.

2.2 Canonical patterns

To extend on the graph mining task described above, we can look for multiple patterns, instead of just one. In this case, one can impose restrictions on the different patterns that are found. For example, it stands to reason that one wants only *canonical* solutions, meaning that no two patterns found are *isomorphic*.

Definition 5. *A graph isomorphism f between two labeled graphs $\mathcal{G} = \langle V, E, l\rangle$ and $\mathcal{G}' = \langle V', E', l'\rangle$ is a* one-to-one *mapping $V \to V'$ such that f represents a homomorphism from $\mathcal{G}$ to $\mathcal{G}'$, and its inverse f^{-1} represents a homomorphism from $\mathcal{G}'$ to $\mathcal{G}$. If there exist graph isomorphisms between $\mathcal{G}$ and $\mathcal{G}'$ we say $\mathcal{G}$ and $\mathcal{G}'$ are* isomorphic.



(a) First candidate pattern

(b) Second candidate pattern

Fig. 2: Possible patterns

Given the graph mining problem as specified in **Fig.** 1, we have already established that **Fig.** 2a is a valid pattern. When we try to mine a second pattern, we might suggest a pattern as shown in **Fig.** 2b. A quick check, however, will show that there is a one-to-one mapping f such that both f as well as its inverse f^{-1} preserve edges. As a result, both candidate patterns are isomorphic, and thus only one should be accepted as a valid pattern.

2.3 Rewording

We want to study how this formal mathematical definition can be expressed in the logics underlying the IDP [De Cat et al., 2016] and the ProB [Leuschel and Schneider, 2014] system. First, we will reword the earlier **Def.** 4 into an equivalent formal definition that uses logical sentences and language constructs available in general logics. In doing this, it becomes evident that the graph mining problem has fundamental underlying characteristics that result in a higher order definition and specification.

The vertices in the graph mining problem have no distinctive property, and can be reused between different example graphs and patterns. Therefore, we will assume one shared, sufficiently large set of vertices V and represent example graphs over these vertices V directly as triples $\langle Edge, Label, Class\rangle$, consisting of an (binary) edge relation on V and a labeling function over V, as well as a classification (positive/negative).

Definition 6. ***Graph Mining (redefined)*** *Given a sufficiently large set of vertices V, a set $\mathbb{G}$ of graphs over this vertex set V, represented by $\langle E, l, c\rangle$ triples where E and l represent the edge relation and labeling function over V respectively, and a template graph $\mathcal{T}$, we look for a graph $\mathcal{P}$ represented by tuple $\langle E_{\mathcal{P}}, l_{\mathcal{P}}\rangle$ such that:*

- $\mathcal{P}$ *is a* connected *subgraph of* $\mathcal{T}$*,*
- $\#\Big\{\langle E, l, pos\rangle \in \mathbb{G} \mid \exists f : f \textit{ is a homomorphism from } \mathcal{P} \textit{ to } \langle E, l, pos\rangle\Big\} \geq N_+$*,*
- $\#\Big\{\langle E, l, neg\rangle \in \mathbb{G} \mid \exists f : f \textit{ is a homomorphism from } \mathcal{P} \textit{ to } \langle E, l, neg\rangle\Big\} \leq N_-$*.*

Definition 7. ***Canonical Patterns*** *A set of* canonical patterns *is a set $\mathbb{P}$ of graphs $\mathcal{P}_1, ..., \mathcal{P}_n$, such that for each pair of different elements (of $\mathbb{P}$) $\mathcal{P}_i, \mathcal{P}_j$ holds that there does not exist an isomorphism between $\mathcal{P}_i$ and $\mathcal{P}_j$.*

Graphs are the main concept in the graph mining problem, and, when represented using triples $\langle E, l, c\rangle$, graphs take the form of *higher order objects*. A set of graphs is equivalent to a set of triples. The most straightforward representation of such a set would be a ternary predicate. As the domains of this predicate range over predicates and functions, it is a higher order predicate.

It is very natural to consider and represent each graph as a *coherent* ensemble of its own components: all characteristics (edges, labeling ...) of a graph are represented by separate entities or concepts, which are grouped together for each graph $\mathcal{G}$ in the triple that describes it. We refer to this as the *local coherence* of the graph representation. Not only is this a very natural representation, this representation also makes it very explicit that all example graphs are *independent*, and that the searches for homomorphisms between a pattern and example graphs are independent as well. This motivates us to reason about graphs as locally coherent objects in our logical models as well. However, the higher order representations needed to reason about graphs and sets of graphs as *coherent* objects in our models are not yet fully supported by the logics of IDP and ASP. In the following section discusses how to solve this using several modeling techniques.

3 Modeling

In this section, we show how state-of-the-art KR systems without support for higher order logic, such as IDP and ASP, can model the graph mining problem and its higher order features using encoding techniques. We identify the desirable properties that from a KR perspective should hold for a good modeling of the graph mining problem and we evaluate how a modeling in ProB, as a KR language with support for higher order sets, satisfies these properties.

3.1 IDP

Existential Second Order The IDP language can express problems that consist of a set of symbols, called the vocabulary V, and a theory, called T, that uses symbols from this vocabulary. The symbols in the vocabulary can be propositions, but they can also represent predicates and functions. These last two types of symbols make the vocabulary, in general, a *second order* object: it is an object that itself *contains* not only propositional symbols, but also first order symbols. For example, vocabulary V in **Listing** 1.1 is a second order vocabulary as it contains the first order symbol `Edge/2`.

The theory T is restricted to a *first order* theory, extended with types, arithmetic, aggregates, and inductive definitions. An example of such a theory is given in **Listing** 1.1. It contains an inductive definition for `Path/2`, and one constraint.

Our inference of choice in the graph mining problem is model expansion; we search for an interpretation I of symbols in the vocabulary V, called a *model*, such that this interpretation I satisfies the theory T. This corresponds to the implicit *existential quantification* of all symbols in the vocabulary, both the propositional as well as the first order symbols. In the example of **Listing** 1.1, we expand the given interpretation S to the model $Result$ with 3 edges: One from the first node to itself, one from the first node to the second, and one from the second to the third. Path contains all corresponding paths between these three nodes.

In conclusion, we say the IDP language can express model expansion for *Existential Second Order* problems. This level of expressiveness is not sufficient for general graph mining problems.

Listing 1.1: IDP example using inductive definitions

```
vocabulary V{
   type Node,
   Edge(Node, Node), Path(Node, Node)
}
theory T : V {
   ∀n[Node] : ∃n2[Node] : Edge(n,n2) ∨ Edge(n2,n).
   {
      Path(x,y) ← Edge(x,y).
      Path(x,y) ← ∃z[Node] : Path(x,z) ∧ Path(z,y).
      Path(x,y) ← Path(y,z).
   }
}
structure S : V{ Node = {1;2;3} }
structure Result : V{
   Node = {1; 2; 3}, Edge = {1,1; 1,2; 2,3}
   Path = {1,1; 1,2; 1,3; 2,1; 2,2; 2,3; 3,1; 3,2; 3,3 }
}
```

Issue 1 First, we must represent the set of example graphs, as specified in **Def.** 6. This definition uses a higher order predicate `GraphInst/3` (See **Listing** 1.2) with the edge predicate as first argument and the labeling function as second argument. For the first graph, `{1,2; 2,1}` and `{1↦ a; 2↦ b}` respectively. It represents a single graph as a tuple of predicates and functions, which is a highly locally coherent representation, preserving the independence of graph characteristics. However, as we are restricted to *Existential* Second Order, we cannot express this higher order predicate in IDP.

One possible solution is to replicate for each graph the different characteristic predicates and functions, as shown in **Listing** 1.3. In this encoding, every graph has its own edge predicate and label function. Because there is now no relation between the different edge predicates and label functions, it is necessary to formulate our theory in terms of these different predicates and functions. Encoding a property such as “In every graph, all nodes have at least two outgoing edges” must be stated for each of the edge predicates explicitly:

```
∀ n[Node] : ∃ n1,n2[Node] : E1(n, n1) ∧ E1(n,n2) ∧ n1 ≠ n2.
∀ n[Node] : ∃ n1,n2[Node] : E2(n, n1) ∧ E2(n,n2) ∧ n1 ≠ n2.
```

It is clear that this solution is undesirable due to the way it scales and the theory modifications needed with growing problem instances. It retains the local coherence and independence of graph characteristics when it comes to data representation, but prohibits the abstraction (generalization) of knowledge in the theory.

Listing 1.2: Higher order predicate modeling the set $\mathbb{G}$ of **Def.** 6.

```
GraphInst({1,2; 2,1},{1↦a; 2↦b},pos).
GraphInst({1,3; 2,1},{1↦c; 2↦b; 3↦a},neg).
```

Listing 1.3: Multiple individual global relations

```
E1(1,2). lb1(1)=a.
E1(2,1). lb1(2)=b.
E2(1,3). lb2(1)=c.
E2(2,1). lb2(2)=b.
         lb2(3)=a.
```

Listing 1.4: Disjoint union using indexed global relations

```
E(g1,1,2). lb(g1,1)=a.
E(g1,2,1). lb(g1,2)=b.
E(g2,1,3). lb(g2,1)=c.
E(g2,2,1). lb(g2,2)=b.
           lb(g2,3)=a.
```

A more workable solution is to represent each characteristic property, such as the edge relation, by a single global relation for all graphs, as shown in **Listing** 1.4. This relation behaves the way it should for a specific graph instance based on an additional argument serving as an identifier for the graph of interest. This global edge relation now corresponds to the *disjoint* or *tagged union* of the graphs' edge relations, where the tags are drawn from a set G consisting of graph identifiers. It is clear that this representation forces us to give up the local coherence of graph characteristics that was present in **Def.** 6. However, generalizing over the different graphs, we can now encode the property stated above as:

```
∀ gid[GraphId] : ∀ n[Node] : ∃ n1,n2[Node] : E(gid, n, n1) ∧ E(gid, n,n2) ∧ n1 ≠ n2
.
```

Issue 2 The homomorphic property can be expressed using a count aggregate, as shown in **Listing** 1.5. First we quantify over all example graphs g, or per *Issue 1*, their identifiers, and subsequently express that there must exist a function f that represents a homomorphism from our pattern graph $\mathcal{P}$ to g.

Listing 1.5: Quantifying over functions outside the vocabulary

$$\#\{g \mid g \in G \wedge \exists\ f\ :\ f \texttt{ is a homomorphism from } \mathcal{P} \texttt{ to } g\} \geq \texttt{N}_{+}$$

However, IDP restricts us to Existential Second Order, which forbids us from quantifying over first order entities such as the function f from **Listing** 1.5 outside of the vocabulary. Thus, we are required to promote the homomorphic mapping functions to a global property in the vocabulary, even though we are only interested in the existence of a mapping, and not in the concrete instance of the mapping itself. We prevent the same explosion of mapping functions as with the graph characteristics in *Issue 1*, by reusing the disjoint union technique proposed above. Note that in this case, the disjoint union technique greatly resembles Skolemization. We introduce a general function `f` that represents all homomorphisms, and make its dependency on a specific example graph explicit using an additional argument: `f(graphId, node):node`. In Second Order Logic, this dependency would follow directly from the syntactic order of the quantifications.

Listing 1.6: Globalized existential functions

$$\#\{g \mid g \in G\ :\ f(g) \texttt{ is a homomorphism from } \mathcal{P} \texttt{ to } g\} \geq \texttt{N}_{+}$$

We can now use this `f` anywhere we would use the regular homomorphic function for a specific graph by fixing the chosen example graph. We denote by $f(g)$ the function f partially applied on argument g. Because the disjoint union technique introduces a single function f which is the union of all these smaller functions, function f becomes partial: it is not defined for tuples where the first the argument is an identifier for a graph $\mathcal{G}$ for which no homomorphic function exists.

Issue 3 The problem of deciding whether a homomorphism from one graph to another exists is $\mathbf{NP}$-complete. As a result, deciding that no homomorphism from one graph to another exists, which forms the basis for the negative homomorphic property, is $\mathbf{coNP}$. As an $\mathbf{NP}$ (or Σ_1^p) solver, IDP cannot solve this problem directly. The straightforward encoding of the negative homomorphic property reuses the result from *Issue 2*:

```
#{g | g ∈ G : f(g) is a homomorphism from 𝒫 to g}≤ N₋
```

But now, our solver must choose a single global function f which satisfies the constraints. It has no obligation to maximize the number of homomorphisms in f, only to satisfy the constraints. Thus, even if there is a negative example $\mathcal{G}_-$ for which a homomorphism exists, the solver can choose f such that f does not represent a homomorphism for this graph $\mathcal{G}_-$. As our constraints are satisfied, we are led to believe that our pattern candidate is a valid pattern.

[Immerman, 1998] has shown that this is inherently linked to IDPs limit to Existential Second Order. Indeed, in order to check that our pattern $\mathcal{P}$ is homomorphic with no more than N_- negative graphs, we have to check that there are enough negative graphs for which no homomorphism exists, for example using a count aggregate as in **Listing** 1.7. By asserting a property for all candidate homomorphic functions f of a certain graph g, the negative homomorphic constraint leads to universal quantification over a function variable.

Listing 1.7: Quantifying over functions outside the vocabulary

```
#{g | g ∈ G ∧ ∀ f : f is not a homomorphism from 𝒫 to g}
```

A way to solve a $\mathbf{coNP}$ problem such as the negative homomorphism constraint using an $\mathbf{NP}$ solver is by encoding the dual (i.e. negated) problem, and conclude that the problem is satisfied if no model exists for the dual problem. This can be checked using an $\mathbf{NP}$ solver. However, this technique can only be implemented in IDP by writing two theories:

- one (positive) theory $\mathcal{T}^+$ (see C), which expresses the positive homomorphic property and generates pattern candidates, and
- one negative theory $\mathcal{T}^-$, which expresses the (dual of) negative homomorphic property and rejects pattern candidates that do not satisfy this constraint.

In IDP, one must provide procedural (lua) code that ties these two theories and their inferences together by allowing the communication of pattern candidates between these two theories.

It is not known whether the problem of graph isomorphism is polynomial time solvable, however it is sure to be no more complex than NP. Conversely, the isomorphism restriction when looking for multiple patterns is also no more complex than $\mathbf{coNP}$. Therefore, we can use the same technique, giving rise to another theory $\mathcal{T}^{iso}$. Note that it is possible to combine the negative theories $\mathcal{T}^-$ and $\mathcal{T}^{iso}$ into a single negative theory.

Inductive Definitions One of the main features of the IDP language is the fact that it extends first order logic with *inductive definitions*. These definitions, evaluated under the well-founded semantics, allow the derivation of negative knowledge that otherwise would be underivable. Take the path predicate defined in **Listing** 1.1. Models of this theory contain the transitive closure `Path`/2 of `Edge`/2. When the edge relation would be chosen such that two nodes a and b are part of two disconnected graphs, there is no model in which `Path(a,b)` holds. Note that when the transitivity property is expressed as an FO constraint instead, there do exist models in which `Path(a,b)` is true.

Other inferences One of the advantages of IDP is its underlying *Knowledge Base* paradigm [De Cat et al., 2016]. Essentially, this paradigm ensures that we can perform other inferences on the graph mining problem. One of these inferences is, for example, optimization. This would allow us to, e.g., minimize or maximize over the number of nodes in the pattern graph, or the number of nodes in the pattern with a certain label, with only minimal changes to the specification.

3.2 ASP

In **ASP**, a language family closely related to IDP, one would mostly encounter the same issues when modeling the graph mining problem. One of the main differences between ASP and IDP is the choice of semantics: ASP looks for the answer set models, whereas IDP looks for well-founded models. Leveraging the minimality property of answer sets, ASP can prevent the invalid models of the example discussed in *Issue 3*. The corresponding technique is called the *saturation* technique [Eiter et al., 2009] and can prevent the creation of two separate theories and writing of procedural code that IDP requires.

When using this technique, ASP detects negative example graphs for which the f does not represent a homomorphism, and requires for these example graphs that f must map every node of the pattern on every node of that example graph, dropping the injectivity constraint. This way, f becomes so large that it is impossible that it belongs to the minimal answer set unless there does not exist a homomorphism from the pattern to this (negative) example graph. Consequently, the minimality property will cause the solver to look for an f that represents a homomorphism for as many example graphs (including negatives) as possible. The same technique can be applied to the isomorphism restriction and other possible Σ_2^p constraints such as subset minimality.

While this technique successfully prevents the need of a procedural loop and the rewriting of the negative homomorphic property and the isomorphism restriction, it is clear that this technique is not derived from a natural KR translation of the Graph Mining definition. Furthermore, as line 1 of **Listing** 1.11 (See C) shows, it is necessary to encode instance specific knowledge into the model.

3.3 ProB

The ProB System can handle mathematical specifications using higher order logic and set theory. As a result, ProB specifications can cover the polynomial hierarchy **PH** [Immerman, 1999].

Higher Order Logic Because of ProB's Higher Order logic support, we can treat graphs as the inherent higher order objects with structure $\langle E, l, c \rangle$ that represents them. This allows us to quantify over a graph and easily access all its characteristic predicates and functions.

ProB's higher order logic support also makes it possible to quantify over the functions f that represent homorphisms locally: there is no need to declare the function f globally, instead they are defined within the context of the set of homomorphic positive (negative) examples. Here, the representation of these functions f is direct, without graph identifier that corresponds to the disjoint union technique as proposed for IDP. Instead, the graph $\mathcal{G}$ for which a homomorphic function is sought, is brought in scope by the quantifier of the set expression.

Because these are now quantified locally, the solver will find a homorphism if one exists, regardless of whether we are expressing the positive or negative homomorphism property. As a result, ProB can model the negative homomorphism property directly, without the need for a second theory and procedural tie-in code.

The same reasoning allows ProB to model the isomorphism restriction when looking for multiple patterns.

Inductive definitions ProB does not support inductive definitions, but allows the expression these constraints using either the B transitive closure primitive or by expressing the completion of the definition. However, these techniques tend to reduce the readability of the constraint, making it difficult for modelers to reason about the connectedness constraint and its derivatives. Furthermore, these constraints incur a high performance loss. Recently, efforts have been made to integrate Kodkod, which provides a high-level interface to SAT-solvers [Torlak and Jackson, 2007], into ProB [Plagge and Leuschel, 2012], which allows offloading these constraints to a SAT-solver that is capable of solving them fast.

3.4 Comparative Summary

Using the graph mining problem as a case study, we derived a set of desirable properties that a good KR specification should satisfy.

1. Labeled graphs are the main concept in the mathematical definition of the graph mining problem. Here, labeled graphs are seen as a mathematical object consisting of an edge relation and a labeling function, and should be treated as higher order objects in the specification.
2. All example graphs are independent, so the search for a homomorphism between a pattern and a given example graph can be performed independently. In essence we want to allow *local* second order quantification.
3. The search for a homomorphism between pattern and example graph is always the same, regardless of the sign of the example graph (negative or positive). The only difference is the at most/at least constraint on the number of homomorphisms. We want a specification that preserves the similarity of these constraints.
4. We want to be able to find multiple, non-isomorphic, patterns.

5. We want to express constraints such as connectedness of the different nodes in the pattern.
6. We want to perform multiple inferences on the problem, with only minimal changes to the model.
7. We prefer a single specification over multiple specifications. Although specifications are preferably modular to make it easier to reuse them, ideally the specification would be solved within a single solver call, requiring no procedural code to tie them together.

Table 1 provides an overview of how the three systems (IDP, ASP and ProB) support the desirable properties, either natively (✓) or using one of the discussed techniques.

Property	IDP	ASP	ProB
1. Graph as a single object	Global disjoint union technique	See IDP	✓
2. Independence of homomorphisms	Global disj. union & partial function	See IDP	✓
3. Similarity of $\geq$ and $\leq$ constraint	Requires theory splitting	No: Saturation technique	✓
4. Multiple patterns (isomorphism)	Requires theory splitting	No: Saturation technique	✓
5. Connectedness	✓	✓	Transitive closure primitive
6. Multiple inferences	Model checking, expansion, minimization	See IDP	Model checking, expansion. Minimization must be encoded in the model.
7. Single solver call	No	One for each pattern	✓

Table 1: Evaluation of the desirable properties in IDP / ASP / ProB for modeling the general graph mining problem based on its key components (matching, pattern enumeration, etc)

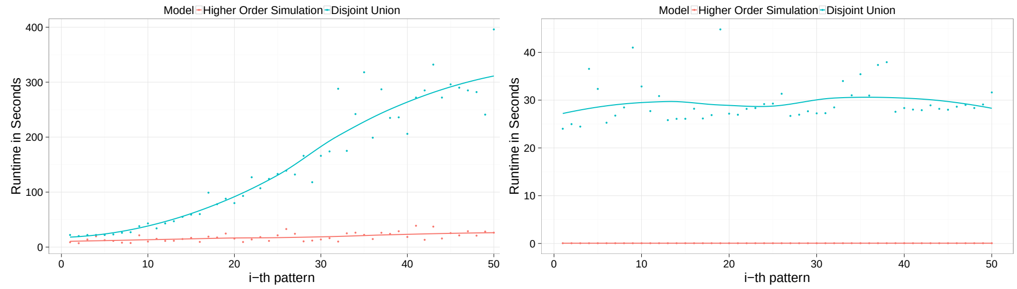
4 Performance

To compare the performance of higher order and first order systems, we compared the IDP system with the ProB system (which uses higher order specifications). To this end, we used the positive examples of the Yoshida [Rückert and Kramer, 2007] dataset, which is derived from biochemics, for graph mining. First, we randomly picked an example to use as the template graph. Next, we mined a pattern from this template, using the threshold value $N_+ = 13$ (5% of the size of the example set). During the mining process, we tracked the time it takes to mine the $i = 1..n$-th pattern. The results are averaged over ten runs.

The ProB model from Subsection 3.3 comes closest to the higher order formulation (as demonstrated in **Table** 1), however, the solver support is not yet sufficient to efficiently execute the higher order graph mining model on larger datasets, i.e., currently we have not found an efficient way to mine patterns using a higher order B model. Consequently, from a KR point of view, we consider the higher order formulation of the graph mining problem as a challenge and goal for future solver techniques. The key issue preventing an efficient higher order formulation lies in reifying the higher order existential quantifier inside the set comprehension. A possible future solution would be to provide a Prolog implementation for the homomorphism predicate (e.g., as a ProB external function). For IDP, the results can be found in **Table** 3.

To analyze the effect of the disjoint union technique, we compared the performance of IDP and ASP on the Yoshida dataset using different encodings of the graph mining problem. In **Fig.** 3, we see the performance of IDP (**Fig.** 3a) and ASP (**Fig.** 3b) on finding the i-th pattern. Two different encodings are used: one that uses the disjoint union technique, and one that performs a new IDP/ASP call for every different example graph, and aggregates this data using procedural code (i.e. in a decomposed fashion).

It is clear from **Fig.** 3 and the order(s) of magnitude difference between the decomposition and disjoint union technique that these systems can highly benefit from detecting the independence of these different subproblems and solving them separately. We expect that expressing the problems in a higher order fashion will allow detection of this subproblem independence and allow for more performant and expressive systems.



(a) IDP: the disjoint model has a growing trend while the simulation stays flat. The gap is one order of magnitude. (Paramonov et al. [2015])

(b) ASP: the disjoint model exhibits fluctuation around 30s with a slow runtime growth, while the simulation stays flat. The gap is two orders of magnitude.

Fig. 3: Frequent graph enumeration problem (5% threshold) on Yoshida dataset for IDP (a) and ASP (b), comparing disjoint union (in blue) and higher order simulations (in red). Further details can be found in A.

5 A faithful encoding

In **Listing** 1.8, we now propose a new encoding for a language combining higher order logic support with the readability of inductive definitions. This encoding is more faithful to the problem with respect to the definition given in **Def.** 6.

In the vocabulary, the second order type `graph`, parametrized by two first order types `node` and `label`, is declared as a tuple of a predicate `vertex/1`, a predicate `edge/2`, and a function `label`. Next, we declare the higher order predicates (`homomorphism`, `reachable`, `isPattern`, `canonical_pattern`, `positive`, and `negative`) and function (`template`).

Within the theory, higher order predicates are defined using the concept of templates as described by Dasseville et al. [2015]. The higher order arguments are decomposed using matching (e.g. line 9) or using dot notation (e.g. line 22). Quantification over second order objects uses annotated quantifiers ($\exists_{SO}$ and $\forall_{SO}$) and must be typed (any unary predicate represents a type), e.g. line 10.

Listing 1.8: Faithful encoding for the general graph mining problem

```
 1  Vocabulary V {
 2    type node, type label
 3    so-type graph(node, label) of (vertex(node), edge(node,node), label(node):label)
 4    homomorphism(graph, graph), reachable(node,node, graph)
 5    isPattern(graph), canonical_pattern(graph)
 6    positive(graph), negative(graph), template:graph
 7  }
 8  Theory T {
 9   {homomorphism((V1, Edge1, Label1), (V2, Edge2, Label2)) ←
10      (∃SO F [V1:V2] : (∀ x, y : x ≠ y  ⟹  F(x) ≠ F(y)) ∧
11       (∀ x, y : Edge1(x, y)  ⟹  Edge2(F(x), F(y))) ∧
12       (∀ x : Label1(x) = Label2(F(x)))).
13   isomorph((V1, Edge1, Label1),(V2, Edge2, Label2)) ←
14      (∃SO F [V1:V2] : (∀ y : y => ∃ x : F(x)=y) ∧
15       (∀ x, y : x ≠ y  ⟹  F(x) ≠ F(y)) ∧
16       (∀ x, y : Edge1(x, y)  ⟹  Edge2(F(x), F(y))) ∧
17       (∀ x, y : Edge2(x, y)  ⟹  ∃ fx, fy : Edge1(fx, fy) ∧ x = F(fx) ∧ y = F(fy))
18       ∧ (∀ x : Label1(x) = Label2(F(x)))).
19   reachable(x, y, (Vertex, Edge, Label)) ← Edge(x, y) ∨ Edge(y, x).
20   reachable(x, y, (Vertex, Edge, Label)) ←∃ z : reachable(x, z, (Vertex, Edge,
         Label)) ∧ reachable(z, y, (Vertex, Edge, Label)).
21   isPattern((Vertex, Edge, Label)) ←
22      ((∀x: Vertex(x)  ⟹  template.vertex(x)) ∧
23       (∀x,y: Vertex(x) ∧ Vertex(y) ∧ template.vertex(x) ∧ template.vertex(y) ∧
             template.edge(x,y)  ⟹  Edge(x,y)) ∧
24       (#{ Pos : positive(Pos) ∧ homomorphism(P, Pos) } ≥ N+) ∧
25       (#{ Neg : negative(Neg) ∧ homomorphism(P, Neg) } ≤ N−) ∧
26       (∀ x, y : reachable(x, y, P))). }
27   ∀P : canonical_pattern(P)  ⟹  isPattern(P).
28   ∀P,P2 : canonical_pattern(P)∧canonical_pattern(P2)∧P≠P2  ⟹  ¬isomorph(P, P2).
29  }
```

This encoding compactly specifies the graph mining problem, in a way that closely corresponds to its mathematical definition. To allow inferences on this theory, extended solver support is necessary. We now propose a way in which a solver can provide this additional support, and potentially even improve performance.

Second order types The solver can represent objects of any `so-type` using the disjoint union technique, declaring a new first order type id containing identifiers for the higher objects, e.g. `graphId`. Using theory analysis, we determine whether the size of the second order type is bounded and if so, impose the same bound on the size of the type id. If no such bound can be detected, we treat id as an infinite type, relying on lazy grounding to create new id objects when necessary and to subsequently instantiate the required rules for the new id object.

Next, every occurrence of an object of type `graph` is replaced by the correct identifier, and quantifications over this type are replaced by quantifications over the set of identifiers. Furthermore, every time a component of an object is accessed (e.g. `Edge/2`) it is replaced by a global predicate representing this component (i.e. `Edge(gid, x, y)`).

Second order quantifications $\exists_{SO}/\forall_{SO}$ Second order quantifications such as $\exists_{SO}$ and $\forall_{SO}$ are supported using the concept of oracles as subsolvers. First, all second order universal quantifications $\forall_{SO} X : \phi$ are rewritten to existential quantification $\neg\exists_{SO} X : \neg\phi$. Suppose now that ϕ does not contain any further second order quantifications. Then the above formula is an existential second order formula, which can be solved by a new instance of the $\mathbf{NP}$solver. Recently, Bogaerts et al. [2016] have identified an interface by which any solver can be nested within another solver. Because our $\mathbf{NP}$solver conforms to this interface, we can modify the $\mathbf{NP}$solver such that it calls a new instance of itself as an *oracle* to evaluate the truth of these formulas. The outer solver is called the *top solver*, and the inner solver is called the *subsolver* or *oracle*. As it is possible to nest these solvers arbitrarily deep, we can now solve a formula of the form $\exists_{SO} X : \phi$, regardless of whether ϕ contains any more second order quantifications. Essentially, the $\mathbf{NP}$solver becomes a $\mathbf{QBF}$solver.

To set up a nested solver for a formula $\exists_{SO} X : \phi$, we must set up a vocabulary V and a theory T over V for this solver. To this end, we first identify the variables Σ used in ϕ. These variables Σ, together with the variable X from the quantification itself, are collected in the new vocabulary V. We call the free variables of ϕ the shared variables Σ_s. We now use the formula ϕ as the theory T for the subsolver.

Whenever the solver needs to evaluate the truth of a second order quantification, the solver simply calls this oracle on vocabulary V and theory T, providing it with a set of assumptions consisting of the values that the top level solver assigns to the shared symbols Σ_s. Depending on whether the subsolver succeeds or fails to find a model, we update the current interpretation of the top solver with the model or learn a new clause, as detailed by Bogaerts et al. [2016]. We expect this subsolver technique to allow detection of the independence of subproblems, thanks to the expressivity of higher order logic, and expect the performance of such a solver to close the gap with the performance of the decomposition technique detailed in Section 4.

6 Conclusion and future work

In this paper we used graph mining as an example of a higher order problem and made a thorough analysis of the problem from the knowledge representation point of view. While techniques exist to express these higher order problems in first order logic, sometimes, explicitly specifying the additional structure HO exhibits allows systems to perform better. For example, in the case of graph mining, higher order logic preserves the local coherence of graphs, and the independence of homomorphisms for the different examples, a property that a higher order solver can exploit in order to raise efficiency. In its current state however, no technique combines the expressiveness of higher order logic with high performance solving techniques.

Inspired by this case study, we propose higher-order language extensions for IDP and propose alternative ways to implement them in the solver. In particular, as shown in Section 4, the use of subsolvers seems promising and will be further explored together with the idea of Benders decomposition [Hooker and Ottosson]. The performance of the encodings in IDP or ASP can be considered as the ultimate target.

Bibliography

H. Abramson and H. Rogers. *Meta-programming in Logic Programming*. MIT Press, 1989. ISBN 9780262510479.

J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996. doi: 10.1017/CBO9780511624162.

J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010. ISBN 0521895561.

B. Bogaerts, T. Janhunen, and S. Tasharrofi. Solving qbf instances with nested sat solvers. 2016.

J. P. Bowen. *Formal Specification and Documentation using Z*. International Thomson Computer Press, 1996.

W. Chen, M. Kifer, and D. S. Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

I. Dasseville, M. van der Hallen, G. Janssens, and M. Denecker. Semantics of templates in a compositional framework for building logics. *TPLP*, 15(4-5):681–695, 2015.

B. De Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, and M. Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312v2, 2016. URL `http://arxiv.org/abs/1401.6312v2`.

L. De Raedt, T. Guns, and S. Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 204–212, 2008.

A. Dries and S. Nijssen. Mining patterns in networks using homomorphism. In *Proceedings of the Twelfth SIAM International Conference on Data Mining, Anaheim, California, USA, April 26-28, 2012.*, pages 260–271, 2012.

T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *IJCAI*, pages 90–96. Professional Book Center, 2005.

T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

M. Gebser, T. Guyet, R. Quiniou, J. Romero, and T. Schaub. Knowledge-based Sequence Mining with ASP. In *IJCAI 2016-25th International joint conference on artificial intelligence*, page 8, New-york, United States, Jul 2016. AAAI.

J. Hooker and G. Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60. ISSN 1436-4646. doi: 10.1007/s10107-003-0375-9. URL `http://dx.doi.org/10.1007/s10107-003-0375-9`.

N. Immerman. Descriptive complexity and model checking. In *FSTTCS*, volume 1530 of *Lecture Notes in Computer Science*, pages 1–5. Springer, 1998.

N. Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

M. Järvisalo. *Logic Programming and Nonmonotonic Reasoning: 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, chapter Itemset Mining as a Challenge Application for Answer Set Enumeration, pages 304–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

M. Leuschel and M. J. Butler. ProB: An automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

M. Leuschel and D. Schneider. Towards b as a high-level constraint modelling language. In Y. Ait Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477 of *Lecture Notes in Computer Science*, pages 101–116. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-43651-6. doi: 10.1007/978-3-662-43652-3_8. URL `http://dx.doi.org/10.1007/978-3-662-43652-3_8`.

L. A. Levin. Universal sorting problems. *Problems of Information Transmission*, 9:265–266, 1973.

J. McCarthy. Elaboration tolerance. In *Working Papers of the Fourth International Symposium on Logical formalizations of Commonsense Reasoning, Commonsense-1998*, 1998.

B. Négrevergne and T. Guns. Constraint-based sequence mining using constraint programming. In *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, pages 288–305, 2015.

S. Paramonov, M. van Leeuwen, M. Denecker, and L. De Raedt. An exercise in declarative modeling for relational query mining. In *International Conference on Inductive Logic Programming, Inductive Logic Programming, Kyoto, 20-22 August 2015*. Springer, Dec. 2015.

D. Plagge and M. Leuschel. Validating b, Z and TLA + using prob and kodkod. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2012. doi: 10.1007/978-3-642-32759-9_31. URL `http://dx.doi.org/10.1007/978-3-642-32759-9_31`.

U. Rückert and S. Kramer. Optimizing feature sets for structured data. ECML '07, pages 716–723, 2007.

E. Torlak and D. Jackson. Kodkod: A relational model finder. In *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–, Washington, DC, USA, 2002. IEEE Computer Society.

A Higher Order Logic Simulation Description

Key dataset characteristics for the experiments, visualized in **Fig.** 3, can be found in **Table** 2.

Table 2: Yoshida datasets parameters

Name	Number of Graphs	Avg Vertices	Avg Edges	Labels	Possible classes
Yoshida	265	20	23	9	2

The experimental setup for the results visualized in **Fig.** 3 is the following: in both disjoint union and higher order models we mined the patterns from smaller to larger in an iterative fashion. First, we set the pattern length, equal to the number of nodes, to two, then computed graph coverage for the pattern. Based on the coverage we add the pattern as frequent and then compute isomorphic patterns in the template. For each isomorphic graph in the template we add a no-good clause. Once all frequent patterns of the length n are mined, i.e., the solver cannot find any other non-isomorphic patterns of the length n, we increase the pattern length to $n + 1$, remove all no-goods and repeat the process.

The key difference between the disjoint union model and the higher order simulation model is in the coverage computation. In case of disjoint union model we make a single call to get a pattern such that it is frequent (i.e., matches at least the threshold amount of graphs) and in the higher order model we make a single call to get a non-isomorphic candidate graph and then a separate call per graph to find if it is covered or not. If we found that a pattern covers more than a threshold amount of graphs, we stop computing the coverage and add the pattern as frequent.

Both models in the described computations follow the general schema used in the specialized algorithms such as gSpan (Yan and Han, 2002). We have also obtained similar runtime patterns on other standard graph datasets described in (Paramonov et al., 2015).

B IDP enumeration results

In this section, we present the experimental results on the general graph mining IDP encoding using theory splitting (that allows incorporating positive, negative examples and other higher order checks in a uniform fashion). We have applied this encoding to Yoshida dataset on positive examples and used the isomorphism check as a negative theory. The results summarized in **Table** 3. The results are consistent with the results in **Fig.** 3 of the more specialized encoding (that uses imperative code around the IDP/ASP calls) based on gSpan schema (Yan and Han, 2002).

C Code

This appendix provides the relevant code for the IDP, ASP and ProB systems. The full IDP code is available at

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Runtime	148	164	173	199	285	364	401	445	490	533	548	585	591	687	802

Table 3: Averaged runtimes in seconds for IDP general graph mining encoding with the theory splitting on the Yoshida dataset.

https://dtai.cs.kuleuven.be/static/krr/files/experiments/aspocp16_IDP.zip and at
https://github.com/SergeyParamonov/LGM,
while the ASP code is available at
https://dtai.cs.kuleuven.be/static/krr/files/experiments/aspocp16_ASP.zip
and the ProB code at
https://dtai.cs.kuleuven.be/static/krr/files/experiments/aspocp16_ProB.zip.

Listing 1.9: IDP positive constraint

```
vocabulary V{
 type node isa nat
 type graphid
 type label

 // Predicates determining the template graph.
 template_edge(node, node)
 template_label(node):label

 // Predicates describing the positive example graphs
 example_edge(graphid, node, node)
 label(graphid, node):label
 threshold: int

 // Predicates describing the pattern graph
 inpattern(node) // True for the nodes which occur in the pattern
 partial f(graphid, node):node // Represents the homomorphisms with the example graphs
 homowith(graphid) // True for graphs for which f represents a correct homomorphism
 path(node, node) // path(a,b): True if there exists a path from a to b in the pattern
}

theory Positive:V_Pos{
  //The pattern is a connected subgraph of the template: From every node in the pattern,
  //there exists a path to every other node in the pattern.
  !x,y[node] : x ˜= y & inpattern(x) & inpattern(y) => path(x,y).
  {
    path(x,y) <- template_edge(x,y) & inpattern(x) & inpattern(y).
    path(x,y) <- ?z[node] : path(x,z) & path(z,y).
    path(x,y) <- path(y,x).
  }

  //existence of a homomorphic f from the pattern to example graph with graphid gid.
  !gid[graphid] : !x[node] : homowith(gid) & inpattern(x) <=> ? y[node] : y=f(gid,x).
  !gid[graphid] : !x,y[node] : homowith(gid) & inpattern(x) & inpattern(y) & x˜=y => f(gid, x) ˜= f(gid,y).
  !gid[graphid] : !x,y[node] : homowith(gid) & inpattern(x) & inpattern(y) & template_edge(x,y) => edge(gid, f(gid,x). f(gid,y)).
  !gid[graphid] : !x[node] : homowith(gid) & inpattern(x) => template_label(x) = label(gid, f(gid,x)).

  // At least N homomorphisms must be found
  #{ gid [graphid] : homowith(graph) } >= threshold.
}
```

Listing 1.10: ASP positive matching

```
0 { homowith(G) } 1 :- positive(G).

1 { f(G,X,V) : node(G,V) } 1 :- positive(G), inpattern(X).

:- used_f(G,X,V1), used_f(G,Y,V2), template_edge(X,Y), not edge(G,V1,V2), inpattern(X), inpattern(Y).
:- used_f(G,X,V), t_label(X,L), not label(G,V,L), inpattern(X).

used_f(G,X,V) :- homo_with(G), f(G,X,V).
:- used_f(G,X,V), used_f(G,Y,V), X != Y.

positive_count(N) :- N = #count{G:homowith(G)}.

:- positive_count(N), N < 13.
```

Listing 1.11: ASP negative matching using saturation technique

```
map(G,X,v1) | map(G,X,v2) | map(G,X,v3) | map(G,X,v4) :- invar(X), negative(G).
map(G,X,V) :- saturated(G), t_node(X), node(G,V).

saturated(G) :- t_edge(X,Y), map(G,X,V1), map(G,Y,V2), not edge(G,V1,V2), negative(G), invar(X), invar(Y).
saturated(G) :- map(G,X,V), map(G,Y,V), X != Y, invar(X), invar(Y). // we cannot map two different template nodes to the same

neg_homowith(G) :- not saturated(G), negative(G).

negative_count(N) :- N = #count{G:neg_homowith(G)}.
:- negative_count(N), N > 1.
```

Listing 1.12: ASP Canonicity template-based check

```
iso(X,x1) | iso(X,x2) | iso(X,x3) | iso(X,x4) :- invar(X).

candidate_var(X) :- iso(_,X).

%not iso!
iso_saturated :- invar(X1), invar(X2), iso(X1,V1), iso(X2,V2), t_edge(V1,V2), not t_edge(X1,X2).
iso_saturated :- invar(X1), invar(X2), iso(X1,V1), iso(X2,V2), not t_edge(V1,V2), t_edge(X1,X2).

iso(X,V) :- invar(X), t_node(V), iso_saturated.

d1(X) :- invar(X), not candidate_var(X).
d2(X) :- not invar(X), candidate_var(X).

not_equal :- d1(X). % check that in fact candidate is different from the pattern itself
not_equal :- d2(X). % check that in fact candidate is different from the pattern itself

iso_saturated :- not not_equal. % should not be completely equal

```

```
min_d1(N) :- N = #min{ X: d1(X) }, not iso_saturated.
min_d2(N) :- N = #min{ X: d2(X) }, not iso_saturated.

iso_saturated :- min_d1(N1), min_d2(N2), N1 > N2.
```

Listing 1.13: ASP auxilary predicates

```
%selects subpattern

t_path(X,Y) :- t_edge(X,Y), invar(X), invar(Y).
t_path(X,Y) :- t_edge(X,Z), t_path(Z,Y), invar(X).

:- invar(X), invar(Y), not t_path(X,Y).

0 { invar(X) } 1 :- t_node(X).
% auxilary constraints


edge(G,Y,X) :- edge(G,X,Y).
t_edge(Y,X) :- t_edge(X,Y).
node(G,Y) :- edge(G,Y,_).
t_node(X) :- t_edge(X,_).
```

Listing 1.14: ASP canonicity previous solution isomorphism check

```
iso(s1,X,x1) | iso(s1,X,x2) :- invar(X).
iso(s2,X,x2) | iso(s2,X,x3) :- invar(X).

candidate_var(G,X) :- iso(G,_,X).

iso_saturated(G) :- invar(X1), invar(X2), iso(G,X1,V1), iso(G,X2,V2), t_edge(V1,V2), not t_edge(X1,X2).
iso_saturated(G) :- invar(X1), invar(X2), iso(G,X1,V1), iso(G,X2,V2), not t_edge(V1,V2), t_edge(X1,X2).
iso_saturatea(G) :- not equal(G), iso(G,_,_).

iso(G,X,V) :- invar(X), t_node(V), iso_saturated(G).

:- not iso_saturated(G), iso(G,_,_).

d1(G,X) :- invar(X), not candidate_var(G,X), iso(G,_,_).
d2(G,X) :- not invar(X), candidate_var(G,X).

not_equal(G) :- d1(G,X). % check that in fact candidate is different from the pattern itself
not_equal(G) :- d2(G,X). % check that in fact candidate is different from the pattern itself

equal(G) :- not not_equal(G), iso(G,_,_).
```

Listing 1.15: ProB specification (without dataset)

```
MACHINE Knowledge
INCLUDES Dataset
SETS
 /* Two predefined sets exist, the vertices that the template and pattern can connect, and the labels.
  * The labels are already defined within Dataset.mch
  */
 Vertices = {x1,x2,x3,x4,x5,x6,x7,x8}
CONSTANTS
 /* The template and our pattern are the constants.
  * * Template is given
  * * Patterns is a set that must be found
  */
  Template,
  Patterns
DEFINITIONS

  SET_PREF_TIME_OUT == 70000; SET_PREF_MAX_INITIALISATIONS == 1;

  /* The (most general, i.e. ternary) definition of homomorphism. Note ’ is the
      property accessor for records*/
  homomorph_with(FromGraph, iso, ToGraph) == (
    iso : Vertices >-> dom(ToGraph’LABEL) &
    !x.( x:Vertices => FromGraph’LABEL(x) = ToGraph’LABEL(iso(x))) &
    !(x,y).( x|->y : FromGraph’EDGES
        => iso(x)|->iso(y) : ToGraph’EDGES)
  );

  /* The (most general, i.e. ternary) definition of isomorphism*/
  isomorphic(FirstGraph, iso, SecondGraph) == (
    #(V1,V2).(
    vertices(FirstGraph’EDGES, V1) &
    vertices(SecondGraph’EDGES, V2) &
    iso : V1 >->> V2 &
    !x.( x:V1 => FirstGraph’LABEL(x) = SecondGraph’LABEL(iso(x))) &
    !(x,y).( x|->y: FirstGraph’EDGES
       => iso(x)|->iso(y) : SecondGraph’EDGES) &
    !(x,y).( x|->y: SecondGraph’EDGES
       => iso~(x)|->iso~(y) : FirstGraph’EDGES)
    )
  );

  vertices(EdgeRelation, Vertices) == (
    Vertices = dom(EdgeRelation) \/ ran(EdgeRelation)
  )

PROPERTIES

  /*This is our given template*/
  Template = {(x1,x2),(x2,x3),(x3,x4),(x4,x5),(x5,x6),(x6,x7),(x7,x8)} &

  /*Typing our Patterns set. It’s a set of records (struct-type) with label a total
       function and edges a relation */
  Patterns : POW(struct(LABEL:Vertices-->Labels, EDGES:Vertices<->Vertices)) &
  /*Derived type: POW(struct(EDGES:POW(Vertices*Vertices), LABEL:POW(Vertices*
      Labels)))*/

  /*A single small test, this is not used anymore but is useful to check edits*/
  /* #isop.(homomorph_with(rec(LABEL:{(x1,a),(x2,b),(x3,a),(x4,a),(x5,a),(x6,a),(x7
      ,a),(x8,a)}, EDGES:{(x1,x2),(x2,x3)}), isop, rec(LABEL:{(1,a),(2,a),(3,b)
      ,(4,a),(5,a),(6,a),(7,a),(8,a)}, EDGES:{(1,2),(2,3),(3,4)},SIGN:"POS"))) &*/

  /* Feed the pattern set with one specific pattern already */
  rec(LABEL:{(x1,a),(x2,b),(x3,a),(x4,a),(x5,a),(x6,a),(x7,a),(x8,a)}, EDGES:{(x1,
      x2),(x2,x3)}) : Patterns &

  /* Requirements on patterns:
   * * The pattern is a subgraph of the template
   * * The number of homomorphisms with positive graphs is great enough (at least-
       requirement)
   * * The number of homomorphisms with negative graphs is small enough (at most-
       requirement)
   * * No two patterns in the Patterns set are isomorphic
   */
  !pattern.(pattern:Patterns => pattern’EDGES <: Template) &
  !pattern.(pattern:Patterns => card({p|p:graphs & p’SIGN="POS" & #isop.(
      homomorph_with(pattern, isop, p))}) >= 1) &
  !pattern.(pattern:Patterns => card({p|p:graphs & p’SIGN="NEG" & #isop.(
      homomorph_with(pattern, isop, p))}) <= 0) &
  !(p1,p2).(p1:Patterns & p2:Patterns & p1 /= p2 => not (#iso.(isomorphic(p1, iso,
      p2)))) &

  #iso.(homomorph_with(rec(EDGES:{(x1|->x2)},LABEL:{(x1|->a),(x2|->a),(x3|->a),(x4|->a),(x5|->a),(x6|->a),(x7|->a),(x8|->a)}),iso,rec(EDGES:{(x1|->x2),(x3|->x4)},LABEL:{(x1|->a),(x2|->a),(x3|->a),(x4|->a),(x5|->a),(x6|->a),(x7|->a),(x8|->a)}))) &

  /* We look for at least n patterns */
  card(Patterns) = 6 &

  1=1
OPERATIONS
 Pat(pattern) = SELECT pattern:Patterns THEN skip END
END
```