The MiningZinc Framework for Constraint-based Itemset Mining

Tias Guns*, Anton Dries*, Guido Tack†, Siegfried Nijssen*‡ and Luc De Raedt*
*Department of Computer Science, KU Leuven
†Caulfield School of Information Technology, Monash University
‡LIACS, Universiteit Leiden
{tias.guns,anton.dries,siegfried.nijssen,luc.deraedt}@cs.kuleuven.be, guido.tack@monash.edu

***Abstract*—We present MiningZinc, a novel system for constraint-based pattern mining. It provides a declarative approach to data mining, where a user specifies a problem in terms of constraints and the system employs advanced techniques to efficiently find solutions. Declarative programming and modeling are common in artificial intelligence and in database systems, but not so much in data mining; by building on ideas from these communities, MiningZinc advances the state-of-the-art of declarative data mining significantly. Key components of the MiningZinc system are (1) a high-level and natural language for formalizing constraint-based itemset mining problems in models, and (2) an infrastructure for executing these models, which supports both specialized mining algorithms as well as generic constraint solving systems. A use case demonstrates the generality of the language, as well as its flexibility towards adding and modifying constraints and data, and the use of different solution methods.**

I. Introduction

The demand for data mining technology is increasing in many organizations. This includes pattern mining technology, which is concerned with finding regularities or patterns in data. However, standard pattern mining algorithms are known to find overwhelming amounts of patterns; too many to be analyzed by the user. A well-known technique to overcome this is to let the user impose additional constraints on patterns [3]. However, the number of constraints supported by existing systems is typically limited to a few built-in and well-studied constraints [2]. For example, formalizing data mining problems that involve multiple data sources at the same time is typically not possible. What is missing is a general language in which constraints on patterns can be expressed, as well as the infrastructure to find the corresponding patterns.

In this paper, we give a demonstration of the current state of the MiningZinc framework, which we are developing to address this challenge. The framework consists of two parts: a language in which pattern mining tasks and constraints can be formalized, and an infrastructure for executing statements in this language. Both parts are closely connected to each other and make the overall system unique.

From the *language* perspective, the key feature of the framework is that it builds on *constraint programming* languages. Constraint programming languages were developed for the high-level specification of constraint satisfaction and optimization tasks. Examples of such languages are Zinc [6] and Essence [4]. The main reason for building on top of these languages is that many constraint-based mining problems are in fact constraint satisfaction and optimization problems, which makes their formalization in these languages rather natural. These languages are very general and their applicability has already been shown in a large number of problems. In particular, our past work has already shown that constraint programming systems can be used in data mining [5]. In this study, we chose to build on top of the Zinc family of languages [6], [7], for which an infrastructure is already available.

From the *infrastructure* perspective, the key feature of the MiningZinc framework is that it supports multiple solution methods. This includes well-known specialized pattern mining algorithms, as well as generic constraint *solvers* developed in the constraint programming community and used in our earlier work [5]. This combination of solution methods is unique, and will allow us to support both a wide range of tasks and constraints, while also supporting the use of highly efficient mining algorithms. For a given problem specification expressed in MiningZinc, our framework can automatically detect which solution methods are applicable, and can execute these.

We built a web interface to promote the interactive use of MiningZinc. It allows a user to load data, either from a file or through an SQL query to a database. The user can then choose a base mining task (or specify one from scratch) and add or modify constraints. Applicable solution methods and the results found are displayed in the same interface. This promotes an iterative approach where the user can add, remove or modify existing constraints or data at any time, and easily compare the resulting patterns found.

The main aim of this paper is to demonstrate the flexible, iterative approach to pattern mining that MiningZinc offers; it will show that within this language it is easy to modify known mining problems, implement new mining problems, and add data sources.

Within this paper, we will first provide an overview of the MiningZinc language; subsequently, we will discuss the architecture of the MiningZinc framework, and illustrate how it can be used in an iterative fashion.

II. The MiningZinc Language

The MiningZinc language can be used to formulate constraint-based itemset mining problems. Itemset mining was introduced by Agrawal et al. [1] and can be defined as follows. The input consists of a set of *transactions*, each of which contains a set of *items*. Transactions are identified by identifiers $\mathcal{S} = \{1, \ldots, n\}$; the set of all items is $\mathcal{I} = \{1, \ldots, m\}$. An

Listing 1. "Frequent Itemset mining"

```
1 int: NrI; int: NrT; int: Freq;
2 array[1..NrT] of set of 1..NrI: TDB;
3 var set of 1..NrI: Items;
4 constraint card(cover(Items,TDB)) >= Freq;
5 solve satisfy;
```

Listing 2. "Cover function in MiningZinc"

```
1 function var set of int: cover(
2                         var set of int: Items,
3                 array[int] of var set of int: D) =
4   let {
5     var set of index_set(D): Trans;
6     constraint forall (t in index_set(D))
7         ( t in Trans <-> Items subset D[t] );
8   } in Trans;
```

itemset database $\mathcal{D}$ maps transaction identifiers to sets of items: $\mathcal{D}(t) \subseteq \mathcal{I}$. The frequent itemset mining problem is then defined mathematically as follows.

Definition 1 (Frequent Itemset Mining): Given an itemset database $\mathcal{D}$ and a threshold *Freq*, the frequent itemset mining problem consists of finding all itemsets $I \subseteq \mathcal{I}$ such that $|\phi_{\mathcal{D}}(I)| > Freq$, where $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$.

The set $\phi_{\mathcal{D}}(I)$ is called the *cover* of the itemset, and the threshold *Freq* the *minimum frequency* threshold. An itemset I which has $|\phi_{\mathcal{D}}(I)| > Freq$ is called a *frequent itemset*.

Minimum frequency is one example of a constraint. Our aim is to develop a language in which on the one hand such constraints can be expressed in a natural way, while on the other hand the language is generic enough to express many other types of constraints. We choose to build on the MiniZinc constraint programming language for this purpose [7]. It is a subset of Zinc [6] restricted to the built-in types *bool, int, set* and *float* and user-defined predicates. The main features of MiniZinc that make it attractive for constraint-based mining is the high-level mathematical syntax, the ability to add user-defined predicates and the independence of the language from solution methods. MiningZinc extends MiniZinc [8] with libraries of functions and predicates that make it easier to implement common constraint-based itemset mining tasks. However, all constraints and primitives present in the generic MiniZinc language can also be used in MiningZinc, making it more powerful than other, specialized declarative data mining systems.

Listing 1 (lines 1 to 5) illustrates the syntax of the language on the frequent itemset mining problem; more examples are given later. Lines 1 and 2 are parameters and data, which a user can provide separate from the actual model. The model represents the items and transaction identifiers in $\mathcal{I}$ and $\mathcal{S}$ by natural numbers (from 1 to NrI and 1 to NrT respectively) and the dataset $\mathcal{D}$ by the array TDB, mapping each transaction identifier to the corresponding set of items. The set of items we are looking for is represented on line 3 as a *set variable* with elements between value 1 and NrI. The minimum frequency constraint is posted on line 4; it naturally corresponds to the formal notation $|\phi_{\mathcal{D}}(I)| \geq Freq$. The *cover* relation used on line 4 and shown in Listing 2 is part of the MiningZinc library and implements $\phi_{\mathcal{D}}(I) = \{t | I \subseteq \mathcal{D}(t)\}$; note that this constraint is *not* a hard-coded constraint in the solver, such as in other systems, but is implemented in the MiningZinc language itself, and can hence be changed if this is desired. Finally, line 5 states that it is a satisfaction problem. Enumerating all solutions that satisfy the constraints corresponds to enumerating all frequent itemsets.

This example demonstrates the appeal of using a modeling language like MiniZinc for pattern mining: The formulation is high-level, declarative and close to the mathematical notation of the problem. Furthermore, the use of user-defined functions allows us to abstract away concepts that are common when expressing constraint-based mining problems, as we will see later.

III. The MiningZinc Infrastructure

The MiningZinc infrastructure is responsible for executing the MiningZinc models. A key design principle is that it should be able to run many different solution methods, as different methods support different kinds of constraints and have different runtime behavior. This allows us to vary the efficiency/expressivity trade-off depending on the model specified by the user: if it only uses standard constraints, one can use highly efficient itemset mining algorithms such as LCM [9], while in the case of many complex constraints one will have to use a constraint solver.

MiniZinc already provides an infrastructure which can be modified to this purpose. We first discuss the basic MiniZinc infrastructure, and then how we extended it.

a) The MiniZinc Toolchain: The MiniZinc toolchain essentially consists of a two-step pipeline with a MiniZinc model and data as input, and the solution given as output. In the first step, the data is added to the high-level solver-independent MiniZinc model and this is compiled into the low-level solver-dependent FlatZinc language. In the second step, a *constraint solver* reads in these predicates and computes the solutions. An optional third step can pretty-print the solution in a way specified by the user.

The constraint solvers that are supported in standard MiniZinc include Constraint Programming (CP) solvers, Mixed Integer Programming (MIP) solvers and Satisfiability (SAT) solvers. Depending on the target solver, it is better to perform different transformations in the compilation of MiniZinc to FlatZinc. For example, a CP solver can handle logical constraints natively, while a MIP solver will want to transform the logical operators into an equivalent integer programming formulation. This can be achieved by overwriting the built-in MiniZinc predicates and operators in a separate *library* file.

b) MiningZinc specificities: We make extensive use of the ability to introduce high-level predicates.

Mining-specific libraries. We added libraries to MiniZinc which define common primitives such as cover and cover_inv. Different libraries provide different instantiations of these primitives. For instance, for some CP solvers it is beneficial to use arrays of boolean variables in the definition of constraints, while for others a set representation works better. This allows the system to use efficient low-level encodings of the problems, while the language remains high-level.

Data Mining algorithms. We also included efficient data mining algorithms in MiningZinc. This is supported through

a model analysis step in the MiningZinc toolchain. This step supports the following:

- it recognizes common formalizations of constraint-based mining problems, including alternative and equivalent formulations, and pairs this with the mining algorithm(s) that support this task;
- in a similar fashion, it can deduce whether a subset of the constraints conform to a common mining task.

The latter feature allows for an automatic hybrid post-processing approach. Once our model analysis concludes that a MiningZinc model can partly be solved using a specialized and efficient mining algorithm, it also recognizes the constraints which are not supported by that algorithm. It then feeds the results of the mining algorithm to a generic constraint solver that can verify whether the remaining constraints are satisfied.

To do the analysis we use the resolution mechanism of a logic programming system. The FlatZinc predicates are transformed into facts, and each mining system has rules that express the combinations of constraints that it supports. As background knowledge we provided a set of possible reformulations of individual constraints, as well as common combinations of constraints. Obtaining the list of supported solvers then amounts to querying each of the rules of the mining systems in turn.

IV. Interface Design & Mining Examples

We integrated the MiningZinc infrastructure in a web-based interface to make it easy to use. A screenshot of this interface is given in Figure 1. In it, data can be loaded, MiningZinc models can be entered, and the results of mining algorithms can be displayed. We demonstrate its functionality by means of a simple example in the analysis of movie data stored in a relational database.

In the first stage of the analysis, the user specifies data sources such as SQL queries to determine the data that will be used in the MiningZinc models. The output of each query is given a name, by which it is accessible in the MiningZinc code. In our example (Figure 1, top), for instance an `ActorsByMovie` transaction table is created in which *actors* are items in transactions of *movies*.

The first idea is to run a frequent itemset mining algorithm to find common combinations of actors. In the input box below "Model" in Figure 1, the model representing frequent itemset mining can be entered. In the model displayed in the screenshot, we also add a *minimum size* constraint to enforce that the itemsets found are of reasonable size. Finally, we add a statement for printing the resulting patterns.

The next step consists of an automatic analysis of the specified model, triggered by pressing the "Analyze" button. In this case, the model analysis concludes that the model requires the specification of two threshold values. The interface dynamically provides appropriate input boxes to specify these (under "Parameters"). After pressing the "Analyze" button again, the system determines which possible solvers can be used to find the requested patterns. There are two categories: the general solvers that support arbitrary MiniZinc models and the specialized data mining solvers. The current infrastructure

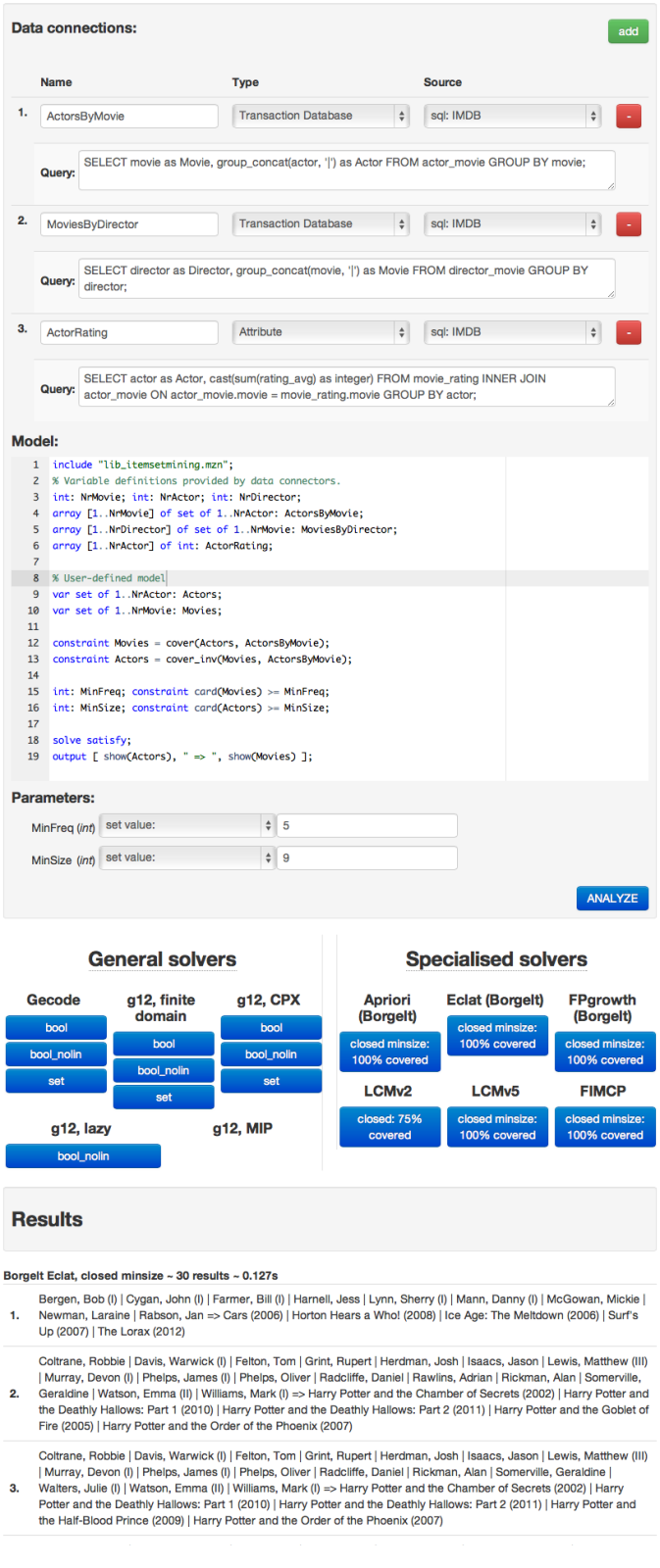

Fig. 1. The web interface of the MiningZinc demo.

supports several different itemset mining implementations, all of which are shown. After the user has selected one of these options, in the bottom of the interface the patterns are shown.

In this example, it can be seen that we mainly find sets

of actors that collaborated in movie sequels; the overall list of patterns found is long. While this is not unsurprising, it may not be what we are interested in. However, by changing the SQL queries and the MiningZinc code it is easy to study alternative mining tasks. Several examples are given below.

Rating-based Constraints. The resulting sets of actors may only consist of relatively unknown actors. We can attempt to focus our attention on well-known actors by including the ratings of movies in our analysis. First, by means of an SQL query, we calculate the average rating of all movies that an actor was involved in (as shown in the screenshot); subsequently, we constrain the search to only those sets of actors with high average rating. By adding the following lines of code in the model this is achieved:

```
1 array[1..Nrl] of int: actor_rating; int: min_rating;
2 constraint ( sum(i in Items) (actor_rating[i]) ) /
3            card(Items) >= min_rating;
```

Director-based Constraints. The most common frequent sets of actors are those that played in sequels of movies for the same director. We may wish to exclude sets of actors that mainly played in such sequels. To obtain this, we can add data about the directors of the movies, and a constraint that requires a minimum number of different directors for selected movies. The following MiningZinc code reflects this and shows that users can extend models with self-defined constraints:

```
1 function var set of int: disj_cover(
2                          var set of int: Items,
3                 array[int] of var set of int: D) =
4   let { var set of index_set(D): Trans;
5         constraint forall (t in index_set(D))
6             ( t in Trans <-> Items intersect D[t] != {} );
7   } in Trans;

8 constraint Directors = disj_cover(Movies,MoviesByDirector);
9 constraint card(Directors) >= FreqD;
```

Discriminative itemset mining. We could limit the number of results further by focusing on itemsets that optimize a given ranking function. One such ranking function is *accuracy* in data with multiple classes of examples. In our movie example, we can obtain two classes of movies by using different SQL queries: one extracts those movies with low ratings and the other those with high ratings. They can be stored in different data matrices (`D_good` and `D_bad`). The following MiningZinc code only searches for sets of actors that maximize the accuracy score on these classes of examples:

```
1 array[int] of set of int: D_good;
2 array[int] of set of int: D_bad;
3 var set of 1..Nrl: Items;
4 constraint Items = cover_inv(
5                cover(Items,D_good),D_good);
6 % Optimization function
7 var int: Score = card(cover(Items, D_good)) -
8                  card(cover(Items, D_bad));
9 solve maximize Score;
```

In line 7 the accuracy scoring function is expressed; line 9 specifies that we only wish to find solutions that optimize the score. Finally, in line 4 we add an additional constraint that limits the attention to *closed* itemsets, which are essentially the longest itemsets that optimize the score.

More. A user can continue analyzing the data by adding constraints and/or data and changing the problem specification.

V. Conclusions

Our demo shows that itemset mining problems can be modeled in a generic constraint modeling language in a natural way. Furthermore, it shows that an infrastructure can be developed which supports both generic and specialized solvers. A web interface offers easy access to the different components: adding data sources, writing the problem specification, choosing a solver and viewing the results.

MiningZinc, as presented here, is only a starting point. Several interesting challenges remain. A first challenge is that of solver selection. Our current infrastructure may recognize which solvers can be applied, but it does not automatically select one. For many end-users this would be beneficial. In general, the current library only supports itemset mining primitives. For a broader applicability in data mining, other primitives should be added, including statistical primitives. From the infrastructure point of view, this would require a wider range of data mining systems to be integrated as well.

Acknowledgements. This work was supported by two Postdoc and one project grant "Principles of Patternset Mining" from the Research Foundation—Flanders, and by the EU FET IST project "Inductive Constraint Programming", contract number FP7-284715. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216. ACM Press, 1993.

[2] Francesco Bonchi and Claudio Lucchese. Extending the state-of-the-art of constraint-based pattern discovery. *Data Knowl. Eng.*, 60(2):377–399, 2007.

[3] Saso Dzeroski, Bart Goethals, and Pance Panov. *Inductive Databases and Constraint-Based Data Mining*. Springer-Verlag New York, Inc., 1st edition, 2010.

[4] Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Essence : A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[5] Tias Guns, Siegfried Nijssen, and Luc De Raedt. Miningzinc: A modeling language for constraint-based mining. *Artif. Intell.*, 175(12-13):1951–1983, 2011.

[6] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia De La Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, September 2008.

[7] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *CP*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[8] Peter J. Stuckey and Guido Tack. Minizinc with functions. In *CP-AI-OR*, volume 7874 of *LNCS*, pages 268–283. Springer, 2013.

[9] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, volume 126 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2004.