Implementation of the Relation Domain for Constraint Programming

Gustavo Gutiérrez[1], Peter Van Roy[2], and Sascha Van Cauwelaert[1]

[1] Université catholique de Louvain, Louvain-la-neuve 1348, Belgium,
{gustavo.ggutierrez,sascha.vancauwelaert}@gmail.com
[2] Université catholique de Louvain, Louvain-la-neuve 1348, Belgium,
peter.vanroy@uclouvain.be

Abstract. Relations are fundamental structures for knowledge representation. Relational queries are used to extract information from associated relations in databases. We propose to include relations in constraint programming (CP) as decision variables and promoting relational operations to constraints. That leads to new possibilities for modeling and solving CSPs. Performance and applicability are two aspects to take into account before considering the new domain practical. In this paper we address the problem of achieving a practical implementation of the system. That includes compact representation of the data in relations and good performance of the relational operations. This compact representation relies upon representing relations into binary decision diagrams (BDDs) and rewriting operations on relations for this specific representation.
Optimized implementations of the operations exist when the BDD representation satisfies particular properties. A solver that maximizes the existence of such properties for a CSP on relations is provided. This ensures the maximum performance from the constraint solver. The implementation of the relation constraint system is publicly available as a Gecode extension[1].

1 Introduction

Relations have been largely used in computer science for diverse purposes that include program analysis [2, 3], data storage [4] and cryptography [5] just to name a few. Data bases are probably the most widely spread example of their use for information representation.

The most appealing feature of relations is the possibility of representing related information by single entities called tuples. Relations can be operated as sets, for instance, by intersecting or complementing them. It is also possible to operate them using specific operations that allow information in tuples to be used. This is done in the relational algebra by operations like projection and join.

Representing and reasoning about information is crucial in constraint programming and relations were proposed as a useful symbolic domain by Gervet in [6]. The idea was revisited in [7, 8] and relations were used in music composition. The significant advantage of using relations is followed by an important

challenge: how to implement them efficiently. However, the expressiveness that relations offer to reason about information in different areas remains undisputed.

This paper presents the implementation of a constraint domain in the framework of a constraint solver. This domain opens the possibility for using relation decision variables to model and solve constraint satisfaction problems (CSPs). When relations are used at the core of a constraint solver, the problem of an efficient implementation becomes more important. Operations on relations take place during the execution of propagators[3].

In the early stages of the implementation we considered the use of data structures like: simple lists of tuples, *Bimaps*[9] and *Multi-index* [10]. Such consideration had two major disadvantages. The use of bimaps only allow representing binary relations. However a common disadvantage of all the data structures is the resulting time complexity of operations that compute the conjunction or complement of a relation.

We found that *Binary Decision Diagrams* (BDDs) have been used for the representation of relations in other fields [11]. Indeed, tools like CrocoPat [11] and Jedd [2], used for software analysis related problems, follow that approach with success. In spite of their theoretical worst-time complexity bounds, BDDs offer efficient representations for a wide range of relations.

Using BDDs to represent relations is not trivial. Particular aspects like variable ordering and choosing the right BDD variables is crucial. Some of those considerations were not taken into account in [7, 8] and constitute improvement. Another consequence of the use of BDDs is that operations like join and composition have optimized implementations that can be used when their argument representations satisfy certain properties. A solver that provides the BDD representation for a CSP problem is also presented.

The result is a system that allows simpler CSP models by allowing a relational formulation of the problem. However, we still require to study the impact of using relations at the solver level and to compare it with other approaches like reformulation or encoding. The implementation is publicly available[1]. It uses Gecode [12] and the CUDD [13] library for the BDD implementation.

1.1 Related work

The idea of a relation domain was proposed in [6]. In [14], relations are used as a powerful modeling mechanism. Functions are particular cases of binary relations and were introduced in [15] to model CSPs. Interestingly, these models were translated into conventional constraint domains (i.e. integers and sets) before being tackled by solvers. The main difference with these works include handling relations of any arity and supporting them in the solver without any translation. Relational algebra has been combined with constraint programming to solve express combinatorial problems in [16], a local search approach is then used to solve the models.

[3] Constraint implementations in the terminology used by Gecode

There have been other structured domains that are related with relations. Graph decision variables were introduced by Régin in [17] and later studied by Dooms (CP(Graph)) [18, 19]. Graphs can be considered binary relations on the set of nodes. That equivalence makes possible the use of the relation domain to handle graphs.

The finite set domain introduced by Gervet and Puget in [20, 21] is also related to relations. A set is equivalent to a unary relation. The similarities between the set and relation domain go further than that. A relation is also a set of tuples on the same schema[4]. Hence, most of the constraints on sets are available on relations. The equivalence between relations and sets will show up several times throughout this paper.

The approach of using BDDs to represent information in constraint programming is not new. Hawkins et al. propose their use to represent set decision variables and set constraints in [22]. That work is related to this one in several forms. First, we use BDDs to approximate the domain of relation variables by using the subset bounds approach introduced in [20, 21]. Second, primitive constraints on sets and integers can be represented as relation values. However, the approach taken by this work focuses on manipulating relations using the operations of the relational algebra. It is important to differentiate the use of decision diagrams as proposed in this work from the work of Andersen et al. in [23]. That work uses Multi Valued Decision Diagrams (a BDD extension) to represent the constraint store. Our proposal is limited to the use of BDDs to represent the domain of relation decision variables.

The approach followed in this work considers two different levels of abstraction depicted in Fig. 1. The bottom level is the relational algebra in which relations and operations like join and projection are defined. The top abstraction introduces the concept of relation decision variables and extends relation operations to have constraints semantics.

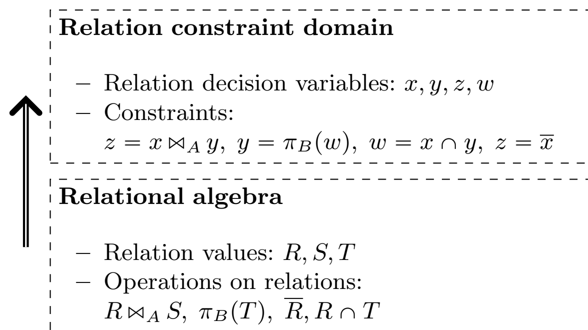


Fig. 1: Abstraction levels considered in this work

[4] The notion of schema is presented in detail in Section 2

1.2 Contributions and plan

The contributions of this paper are:

1. A formal definition of relation decision variables. That is, variables that take relations as values. Basic constraints on this new kind of variables are also introduced and defined.
2. A discussion on one possible representation of the domain of relation decision variables using BDDs along with the obtained benefits and drawbacks.
3. The design and implementation details of constraint decision variables in the Gecode library using the CUDD library for the BDD manipulation.

This paper is structured as follows. Section 2 presents preliminary definitions from the relational algebra and the operations required to understand the subsequent sections. In Section 3 the relation constraint domain and the domain approximation is presented. Section 4 presents how relations and relation decision variables are represented using BDDs. Sections Sections 5 and 6 present important considerations that derive from the use of BDDs in this work. Finally, Section 7 presents the conclusions and the foreseeable future work.

2 Preliminaries

This section provides some reference definitions from the relational algebra and correspond to the abstraction level presented at the bottom of Fig. 1. Notions like schema, relations and join were introduced by Codd in [4].

Definition 1. *Relation Schema [4]. A relation schema* $s = r(a_1 : D_1, \ldots, a_n : D_n)$ *consists of a relation name* r *and a set of attributes* $a_1, \ldots, a_n$ *associated with the domains* $D_1, \ldots, D_n$ *respectively. We refer to the domain of the attribute* a_i *by* $dom(a_i)$ *and to the set of attributes of* s *by* $Attributes(s)$*.*

Definition 2. *Relation [4]. A relation* R *on a schema* $r(a_1 : D_1, \ldots, a_n : D_n)$ *is a set of tuples. A tuple* $t = \langle a_1 : D_1, \ldots, a_n : D_n \rangle$ *is a function from* $Attributes(r)$ *to values in their respective domain.* $t(a_i)$ *is the value for attribute* a_i *of* t*. All the tuples in a relation have the same attribute set.*

Regarding tuples as functions has the advantage of making the order of columns in a relation immaterial. We use $Schema(R)$ to refer to the schema of R. Depending on the number of attributes in its tuples, a relation is said to be *unary*, *binary* or *n-ary* for one, two or any number of attributes respectively.

An empty relation contains no tuples. A full relation contains all the tuples that can be formed from its schema. The set of attributes in the schema can be empty and in such case the relation has only the empty tuple as possible elements.

Operations on relations can be classified in attribute and tuple operations depending on their granularity. Tuple operations treat relations as sets of tuples. Examples of this kind of operations include union, intersection, etc.. Attribute

operations allow finer granularity and work at the attribute level. Projection and join are representatives of this group.

Set operations require their operands to be on the same schema. The complement of a relation R uses the full relation on $Schema(R)$ as reference set. Set operations are not defined for space reasons. The following auxiliary definitions are used to define attribute operations.

Definition 3. *Tuple restriction. Given a schema $r(d_1 : D_1, \ldots, d_n : D_n)$ and a tuple $t = \langle d_1 : D_1, \ldots, d_n : D_n \rangle$ on that schema; we denote by $t[A]$ the restriction of the tuple t on the set A of attributes. The result is a tuple which contains only the attributes of t that are in A.*

$$t[A] = \langle a_1 : D_1, \ldots, a_m : D_m \rangle \iff A \subseteq Attributes(r) \wedge \\ \forall a_i \in A : t(a_i) = t[A](a_i)$$

Definition 4. *Tuple matching. $t \#_M u$ is a predicate evaluating to true if and only if tuples t and u have the same values for the attributes in M.*

$$t \underset{M}{\#} u \iff \forall_{a \in M}\ t(a) = u(a)$$

With the above definitions we present formal definitions for projection and join of relations.

Definition 5. *Projection. Projecting a relation R on a set of attributes A results in a relation containing the tuples in R after removing the attributes not in A. Formally:*

$$\pi_A(R) = \{t[A] : t \in R\}$$

The removal of an attribute from two tuples can result in the same tuple. As relations are sets no repeated elements are represented. For that reason, projecting a relation may result in a relation with less tuples. Moreover, for the operation to be well defined $A \subseteq Attributes(R)$ is required.

Definition 6. *Join. Let R and S be relations on possibly different schemas. Joining R and S with respect to a set of attributes M results in the relation:*

$$R \underset{M}{\bowtie} S = \{t : \ \exists r = t[A_R],\ \exists s = t[A_S] \ \wedge r \in R \wedge s \in S \wedge r \underset{M}{\#} s\}$$

Where $A_R = Attributes(R)$ and $A_S = Attributes(S)$ and $M \subseteq A_R \cap A_S$. The set of attributes of the resulting relations is $Attributes(R) \cup Attributes(S)$.

3 Relation constraint domain

This section uses the concepts presented in Section 2 to define the relation constraint domain which is one of the contributions of this work. This part corresponds to the top level of abstraction presented in Fig. 1.

Definition 7. *Relation decision variable. A relation decision variable (or relation variable for short) is assigned a relation and its domain consists of the set of possible relations that the variable can be assigned to. All the relations in the domain of a relation variable have the same schema.*

As done for other domains in [24], the constraint domain on relations is defined on a set D of relations on possibly different schemas and the set $\Sigma = \{=, \in_{[r,s]}, \cap, \overline{r}, \bowtie, \pi\}$ of operations on relations. For any pair of relations r and s, such that $Schema(r) = Schema(s)$, $r \cap s$ denotes the relation resulting from intersecting r and s; similarly, $r \cup s$ represent their union. The complement of a relation r is represented by $\overline{r}$. Additionally, given a relation t on a (possibly different) schema than r, $r \bowtie_A t$ is the relation resulting from joining r and t on the possibly empty set A of attributes. $\pi_A(r)$ is the relation resulting from the projection of r on the set A of attributes. $t \in_{[r,s]}$ is the predicate that evaluates to true if and only if $r \subseteq t \subseteq s$.

Let $\mathcal{D}$ interpret the symbols in Σ as the usual relation operations as defined above. Now, we define $\mathcal{L}$ as the set of primitive constraints on relations. Let x, y and z be relation decision variables on the same schema. $x = y$ denotes the equality constraint, $z = x \cap y$ is the intersection constraint, $y = \overline{x}$ is the complement constraint. Let w and z be relation decision variables on (possible) different schemas from x and y. $z = x \bowtie_A w$ is the join constraint and $z = \pi_A(x)$ is the intersection constraint. The relation constraint domain is defined as $(\mathcal{D}, \mathcal{L})$.

The existence of operations like join and projection makes possible to have relations on different schemas in D. Those operations result in relations on different schemas than their arguments. The composition operation on relations is missing from Σ. The reason is that composition is not considered a basic operation because it can be achieved by a combination of join and projection.

3.1 Domain representation for a relation variable

The domain of a relation variable is a set of relations on the same schema. To get the constraint domain implemented we require to represent such set. Moreover, such implementation needs to be efficient in terms of space and time because propagators use and update that information.

An extensional representation of the domain is not space efficient, it is potentially costly and therefore non practical. Hence, we have to consider the alternative of approximating it. To that end we use an interval representation.

Definition 8. *Relation interval. A relation interval is denoted by* $[r, s]$ *and represents the set of relations that have* r *included and that are included in* s*. Formally it corresponds to the set:* $\{x : r \subseteq x \subseteq s\}$*.*

Let $s = r(d_1 : D_1, \ldots, d_n : D_n)$ be the schema of a relation in the constraint domain. Let F_s be the full relation on schema s. The set $\mathcal{P}(F_s)$ is partially ordered by relation inclusion. Moreover, $(\mathcal{P}(F_s), \subseteq)$ is a lattice with $\varnothing$ and F_s as bottom and top elements respectively [25]. The join operation of the lattice is

union while the meet operation is intersection. This is exactly the same representation used for set variables in [6], but with tuples instead of simple integers as set elements.

Due to the lattice structure it is possible to approximate the domain of a relation variable by a relation interval $[r, s]$. r is the greatest lower bound (glb) and s is the least upper bound (lub). For a relation variable x with the domain approximated by the relation interval $[l, u]$ we say that $l = glb(x)$ is the known relation and $u = lub(x)$ is the possible relation. The information represented by the domain can be changed by growing the lower bound and by decreasing the upper bound.

3.2 Primitive constraints

In the previous section we introduce the relation constraint domain. This section presents the join and projection constraints for relation variables. These two constraints are part of the primitive constraints of the domain. Constraints with set semantic are omitted for the lack of space.

For each constraint we define its semantics and the filtering rules. Those rules represent the changes on the bounds that are required to enforce the semantics of the constraint. The left side of the rule has the bound being affected and the right side contains the expression representing its new value. The appearance of a bound on the right side refers to its current value. Projection and join constraints are defined in terms of projection and join operations on relations. When the symbols π and $\bowtie$ appear on the filtering rules they denote operations on relation values.

Definition 9. *Projection constraint. Let x and y be two relation variables on possibly different schemas. The projection constraint holds if y is the projection of x over the set of attributes A; denoted: $y = \pi_A(x)$. For the constraint to be well defined $Attributes(y) \subseteq Attributes(x)$ and $A \subseteq Attributes(x)$.*

The filtering rules enforcing the constraint on the domains of its variables are:

1. $glb(y) \leftarrow glb(y) \cup \pi_A(glb(x))$.
2. $lub(y) \leftarrow lub(y) \cap \pi_A(lub(x))$.
3. $lub(x) \leftarrow lub(x) \cap (lub(y) \times R_F)$
4. The constraint is failed if $glb(x) \nsubseteq (lub(y) \times R_F)$
5. $lub(x) \leftarrow (glb(y) \times R_F) \cap (\exists!_A lub(x))$

R_F is a full relation on the schema $s(Attributes(x) \backslash Attributes(y))$. Computing the Cartesian product of R_F with the bounds of y produces relations on the same schema of x. $\exists!_A$ is the unique operation. Its application results in a relation with tuples on x that are unique when only the attributes in A are considered.

Definition 10. *Join constraint. Let x, y and z be three relation variables on their respective schemas and A be a set of attributes. The constraint holds if z*

is the result of joining the relations x and y on the set of attributes A, denoted by $z = x \bowtie_A y$.

For the constraint to be well defined it is required: $Attributes(z) = Attributes(x) \cup Attributes(y)$, $A \subseteq Attributes(x)$ and $A \subseteq Attributes(y)$.

The filtering rules for the join constraint are:

1. $glb(z) \leftarrow glb(z) \cup (glb(x) \bowtie_A glb(y))$
2. $lub(z) \leftarrow lub(z) \cap (lub(x) \bowtie_A lub(y))$
3. $glb(x) \leftarrow glb(x) \cup \pi_{A_x}(glb(z))$
4. $lub(x) \leftarrow lub(x) \backslash \pi_{A_x}(glb(y) \bowtie_A \overline{lub(z)})$
5. $glb(y) \leftarrow glb(y) \cup \pi_{A_y}(glb(z))$
6. $lub(y) \leftarrow lub(y) \backslash \pi_{A_y}(glb(x) \bowtie_A \overline{lub(z)})$

The similarities between join and intersection are also appreciated in the filtering rules. Apart from the projection operation used, the rules are mostly the same as the ones for the set intersection constraint. Projection is used to achieve set compatibility.

4 Relation representation using BDDs

The goal of using BDDs to represent relations is efficiency. Operations on relations are performed by propagators. For that reason, we need time- and space-efficient data structures. BDDs provide a way to represent boolean functions, they were introduced in [26, 27] but were made practical by the work of Bryant [28]. The have been also used in the representation of other types of decision variables, such as sets in [22].

To describe the way we use BDDs to represent relations we first need to introduce the concept of a BDD. We only introduce the operations on BDDs that are important to that end. For a more detailed information on this data structure the reader is referred to [28].

4.1 Binary decision diagrams

A BDD is a rooted directed acyclic graph which is derived by reducing a binary decision tree. A binary decision tree has decision nodes, 0-terminal nodes and 1-terminal nodes. Each decision node is labeled with a boolean variable and has two children, called low child and high child [3]. We restrict ourselves to the case in which each decision variable appears at most once in any path from the root to a terminal node.

A decision tree represents a relation of the form $\{0, 1\}^n \rightarrow \{0, 1\}$. The inputs are the values of the decision variables. The output is the terminal node reached by traversing the tree from the root and following the children according to the input values. If the value of the input variable i is 1 then the path continues with the high child and otherwise the low child is used.

Decision trees have an exponential worst-case space complexity. That can be reduced in some cases by transforming the tree into a BDD. Such transformation is made by a bottom-up application of the following rules [3]:

– Merge any isomorphic subtrees.
– Eliminate any node whose low child and high child are identical.

The application of these rules lead to an acyclic graph which is the BDD. Moreover, by fixing the order which the variables (represented by decision nodes) appear along a given path in the graph guarantees a canonical representation of the relation [28]. It is worth noticing that the potentially exponential complexity still remains. However, in many cases BDDs have proved effective to produce compact representations [29].

4.2 Mapping relations to BDDs

Representing a relation by a BDD is the core of the constraint domain implementation. A relation interval is then represented by two BDDs just like the subset bounds representation of sets proposed in [22]. Propagators interact with the bounds to represent new information.

A BDD is the representation of a function F that maps elements from $\{0,1\}^n$ to $\{0,1\}$. The inputs of F are tuples in $\{0,1\}^n$ and its valuation indicates whether the tuple is a member of the function. Representing a tuple $\langle d_1 : D_1, \ldots, d_n : D_n \rangle$ directly on a BDD is not possible because the attribute values are not binary. To solve this we use the binary encoding of each d_i instead.

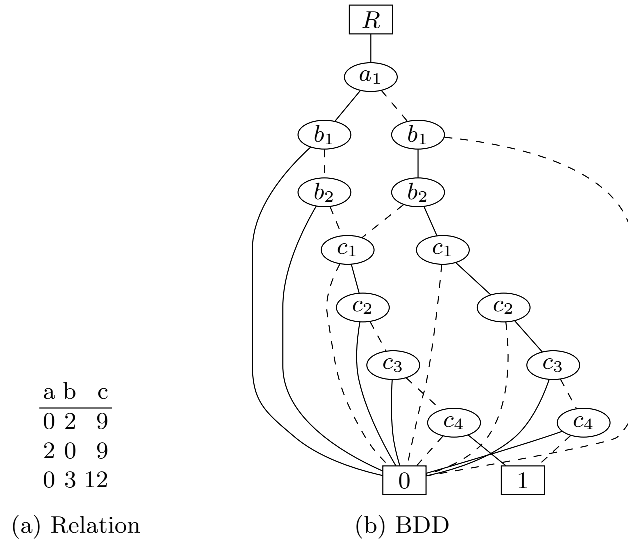
Example 1. Consider three attribute domains $A = \{0,1\}$, $B = \{0,\ldots,3\}$ and $C = \{0,\ldots,15\}$ and the schema $r(a : A, b : B, c : C)$. We require 1 BDD variable to represent the attributes in A, 2 for the attributes in B and 4 for the ones in C. Figure 2 presents relation R on the above defined schema and its corresponding BDD representation[5]. Each level in the figure represents a BDD variable. a_1 encodes the attribute a while $\{c_1, c_2, c_3, c_4\}$ encode the attribute c. Paths from the root node R to terminal 1 encode the elements of the tuples that belong to the relation.

To appreciate how a tuple is encoded in the BDD of Fig. 2, consider the following path:

$$a_1 \dashrightarrow b_1 \longrightarrow b_2 \longrightarrow c_1 \longrightarrow c_2 \longrightarrow c_3 \dashrightarrow c_4 \dashrightarrow 1$$

Tuple values are encoded in the edges. A dashed edge represents a value of 0 for the variable represented by the source node while a solid one represents a value of 1. The path above represents $a_1 = 0, b_1 = b_2 = c_1 = c_2 = 1, c_3 = c_4 = 0$. The encoded tuple, still in binary format is $\langle a : 0, b : 11, c : 1100 \rangle$ which converted to decimal[6] is $\langle a : 0, b : 3, c : 12 \rangle$.

[5] The picture shows the terminals 0 and 1 for readability purposes. In practice, only one terminal needs to be represented if the implementation uses complemented edges.
[6] Assuming the most significant bit is on the left



a	b	c
0	2	9
2	0	9
0	3	12

(a) Relation (b) BDD

Fig. 2: Example showing a relation on schema $r(a:A, b:B, c:C)$ and its corresponding BDD representation.

4.3 Relation operations as BDD operations

After representing a relation with a BDD we define how operations on relations are performed in terms of operations on their BDD representations. One of the advantages of using BDDs to represent relations is that we perform relation operations directly on the BDD representation.

In Section 3 we defined $\Sigma = \{=, \in_{[r,s]}, \cap, \overline{r}, \bowtie, \pi\}$ as the set of operations on relations of our interest. In this section we present how each of these operations is performed on relations represented as BDDs. We leave out the representation of the operation $\in_{[r,s]}$ as it is used for the declaration of relation decision variables and is enforced by construction of the relation interval.

- $R = S$: Equality test is performed by testing if the representations are equal.
- $R \cap S$: Conjunction of the BDDs representing R and S.
- $\overline{R}$: Negation of the BDD representing R.
- $\pi_A(R)$: Existential quantification of the BDD variables of the BDD representing R that are used to represent the attributes in A.
- $R \bowtie_A S$: Conjunction of the BDDs representing R and S.

Both intersection and join operations are computed with the conjunction of the BDDs representing the argument relations. That is because intersection is just an special case of join where both relations have the same schema and therefore are represented on the same BDD variables. How the schema is used to determine which BDD variables must be used to represent a given relation is explained in Section 4.4.

To compute the join of two relations R and S on an attribute set A, suppose that the following properties are satisfied by the representations of the relations:

- The attributes in A of R and S are represented on the same set of BDD variables.
- The only BDD variables shared by the representations of R and S are the ones used to represent attributes in A.

When the properties above are satisfied then computing the join between R and S is equivalent to computing the conjunction of their respective representations. However, This can not be the general case. It does happen that sometimes R and S share the representation of more attributes than one would be interested in a particular join operation and therefore second property is not satisfied. Moreover, it can happen that there are two columns one would like to join that are not represented on the same BDD variables and hence, first property is not respected.

Before computing the conjunction we have to create intermediate representations of R and S that satisfy the properties. That is done using an operation that copies a BDD into other BDD but using a different set of variables. After that the join is computed as described above and an extra copy step is required to copy back the result into the original representation. In this case, the complexity of the join operation is incremented by the complexity of the two copy operations involved. For that reason it is very important to carefully choose how the relation schemas of the relation decision variables in a given CSP are represented in terms of BDD variables. Such a selection needs to take into account not only the relations themselves but the operations that will be performed on them. For that reason we propose to use a constraint solver for the schema allocation problem Section 6.

Finally, this problem also affect the composition operation. Composition of two relations R and S on a set of common attributes A can be computed by a join operation followed by the existential quantification of the BDD variables representing attributes in A.

4.4 Schema representation

Every relation has an associated schema that is taken into account in all the operations. A schema constrains the values that a given attribute can take on. Using that information we compute how many BDD variables are needed to represent the attribute data. Additionally, the attributes of the same schema need to be represented by different BDD variables.

The schema is also used to decide which BDD variables are selected for the data representation. Such decision impacts all the relations on those schemas causing the performance and complexity of the operations on them to be indirectly affected. For example, consider a join that fulfills the requirements for implementation by a single conjunction against one that require the representations to be copied.

Moreover, the decision is different for every model because the operations on the relations depend on the constraints of the problem. Actually, the problem of allocating BDD variables for schema representations is NP-hard. Our approach follows the initiative of Jedd [2] but we use constraint programming to tackle it.

Assigning BDD variables for schema representation does not affect the correctness of the operations but their performance. The more copy operations there are, the (potentially) greater space and time the operations on relations take up. The problem is exacerbated by the fact that the operations are performed at the constraint implementations. For instance, the join propagator performs three join operations per execution. For that reason we create the model for the *schema allocation problem* that is explained in Section 6.

5 Implementation considerations

The relation domain has been implemented as an additional constraint system of Gecode [12] using the BDD implementation CUDD [13]. Using BDDs for the representation of relations requires some details to be considered. The memory management of CUDD uses reference counting[7]. The disposal mechanism of variable implementations in Gecode is used to interact with the memory model of the BDD implementation.

Representing the two bounds of a relation interval is optimized at the BDD level. In an interval, the lower bound is a subset of the upper one. Moreover, both bounds are relations on the same schema and therefore their representations use the same set of BDD variables. That makes the two BDDs to have common parts. Those parts are represented only once and shared at the implementation level.

Gecode uses copy as the basic mechanism for space restoration. Using BDDs for relation representation interacts nicely with that approach. The reason is again sharing of BDDs. Copying a variable during search is a constant time operation and will reuse the BDDs used by the variable in the parent space when possible.

One extra operation that is worth mentioning and that benefits the performance of the implementation is constant time complementation. Computing the complement of a relation requires no extra space and is performed in one instruction in BDD implementations like CUDD that use complemented edges.

It is important to insist in the complexity of the operations on BDDs. In theory the number of nodes in a BDD can be exponential in the number of variables in the represented function. Moreover, the complexity of the BDD operations are given in terms of the number of internal nodes of the operands. In our case, BDD operations get executed very often because they are part of the propagators. In general, however, BDDs offer compact representation for structured data and therefore the theoretical complexity does not constitute the average case.

[7] This is a general aspect in most BDD implementations.

Finally, an important consideration that has to be measured is the variable ordering of the BDDs. Also the impact of reordering algorithms during the solving phase has to be evaluated. The use of CUDD makes this order global for all the representations.

6 Schema allocation problem

The schema allocation problem aims at finding an assignment of BDD variables to attribute domains. Given a CSP that uses the relation constraint domain we need a constraint solver that finds a set of BDD variables to each attribute domain of each schema.

To simplify the model we assume that there are enough BDD variables to represent all the relations introduced in the CSP. With this assumption we simplify the problem to finding an assignment of relation attributes to identifiers. Each identifier represents a set of BDD variables. The simplification is sound because the complexity in a BDD implementation resides in the number of nodes and not in the amount of BDD variables.

6.1 CSP model

The input to the problem consists of a description of the relations and the operations among them. This input is represented by two constants in the model: *Relations* and *Operations*.

To model the problem we use an array of integer decision variables *Column*. Each variable in the array corresponds to a column of a relation in the input. The initial domain for all the variables in *Column* is the set of available identifiers.

Relations in the model are connected by the operations. An operation might require the copy of representations of some of its attributes. This is modeled by using a matrix of set decision variables called *Rename*. Rows and columns of the matrix are labeled by relation columns in the input. The elements of the sets are operations in the input. An operation *op* belongs to the set $Rename(i, j)$ if such operation requires a renaming (i.e. a copy) of columns i and j.

Finally, the integer decision variable *Renames* is used for optimization. It contains the number of rename operations required by a given solution. This variable will be optimized by the search engine.

The following is the complete specification of the CSP model.

Constants:

- *Relations*: Set of relations in the input.
- *Operations*: Set of operations in the input.
- *numColumns*: Total number of columns in relations in the input.

Variables:

- *Column*: Array of integer variables of size *numColumns*.
- *Rename*: Matrix of set variables, size: $numColumns \cdot numColumns$.
- *Renames*: Integer decision variable.
  Total of rename operations required.

Initial Domains:

$\mathit{Column}(i)$	$\{1, \ldots, \mathit{numColumns}\}$.
$\mathit{Rename}(i,j)$	$\mathcal{P}\left(\mathit{Op}_{ij}\right)$. $\mathit{Op}_{ij} \subseteq \mathit{Operations}$: operations in which columns i and j are involved.
$\mathit{Renames}$	$\{0, \ldots, r\}$ $r = \sum_{i=0}^{n} \sum_{j=0}^{n} \lvert \mathit{Rename}(i,j) \rvert$ $n = \mathit{numColumns} - 1$.

Constraints:

- Columns of the same relation need to be allocated to different set identifier. For every relation R in the input, let C be the set of columns of R:

$$\forall i \in C, j \in C : i \neq j \implies \mathit{Column}(i) \neq \mathit{Column}(j)$$

- Set operations require relations with the same schema. For every operation $\langle \mathit{kind} : \mathit{set}, \mathit{id} : I, \mathit{lhs} : L, \mathit{rhs} : R, \mathit{relates} : M \rangle$ post the constraint

$$\forall (i,j) \in M : \mathit{Column}(i) = \mathit{Column}(j)$$

- Join and composition operations require paired columns to be assigned to the same set identifier. Not involved columns need to be assigned different identifiers. For every operation $\langle \mathit{kind} : K, \mathit{id} : I, \mathit{lhs} : L, \mathit{rhs} : R, \mathit{relates} : M \rangle$, with $K \in \{\mathit{join}, \mathit{compose}\}$ post the constraint

$$\begin{aligned}
&\forall (i,j) \in M : \\
&\quad I \in \mathit{Rename}(i,j) \iff \mathit{Column}(i) \neq \mathit{Column}(j) \\
&\forall (i,j) \in (C_L \times C_R) \backslash M : \\
&\quad I \in \mathit{Rename}(i,j) \iff \mathit{Column}(i) = \mathit{Column}(j)
\end{aligned}$$

- Optimization is performed based on the number of rename operations.

$$\mathit{Cardinality}\Big(\bigcup_{i,j \in \{0,\ldots,n\}} \mathit{Rename}(i,j)\Big) = \mathit{Renames}$$

Optimization:

Let $b = \mathit{Renames}$ be the minimum number of renaming operations used so far. The following constraint tries to find an smaller number of renames.

$$\mathit{Renames} < b$$

The model above is implemented in Gecode. A post processing of the solution is required to change set identifiers by BDD variable indexes. Copy operations can be taken into account by constraint implementations directly. Another option is to provide an implementation of the equality constraint on relations that handles the equality property through a mapping of attribute domains. The later has the advantage of simpler constraint implementations at the cost of superfluous variables and constraints in the model.

7 Conclusions

We have presented the relation constraint domain and how to implement it in an efficient way. Such efficiency is achieved by using BDDs to represent relation values. Even tough we use BDDs as means to implement relations, representing them has exponential complexity in the worst case. However, that depends on the particular problem data. Even in the case of large BDDs the advantages described in Section 5 are still useful.

Music composition has been so far the only application of the relation domain. Fundamental improvements from the first stages of the domain implementation have been made [8]. Such improvements have been possible by the real use of the domain in the musical field. Actually, music composition has demanded new constraints [6] that are implemented but were not described for the lack of space.

Constraint solving using relations and structured domains have proved to be a rich and challenging field. Graphs [19], functions [15] are examples of that. One challenge posed by symbolic domains implementation is the efficiency of the domain representations. This work uses the most efficient implementation of relations that we are aware of. That is ratified by the use of BDDs to represent information in program analysis and VLSI design.

Future work

Improvements of the constraint domain can go in several directions. We have seen a lot of feedback by using it to model problems in music. We think there are still many areas in music that can benefit from the use of this system. Relational reasoning has been used in program analysis and that field may offer diverse places for the use of constraint programming and in particular of the relational domain.

Our next step is to use the relation constraint domain in the context of safe capabilities pattern analysis [30]. In concrete, we are interested in studying SCOLL and its implementation [31]. The use of constraint programming in that work uses a lot of relation reasoning.

It is important to notice that presenting a comparison of this implementation is difficult because we are not aware of a similar relation domain. As future work we have also considered comparing how the implementation compares to the domains mentioned in Section 1.1.

The introduction of BDDs was followed by different generalizations. For instance, Algebraic Decision Diagrams (ADDs) and Edge-Valued Decision Diagrams (part of a class called Multi-Valued Decision Diagrams). So far we have only used BDDs but it still remains open the use of MDDs a careful study is also required in this direction.

References

1. Gutiérrez, G.: CPRel: Constraint Programing on Relations (https://bitbucket.org/ggutierrez/cprel4). (June 2012) MIT Software license

2. Lhotak, O., Laurie, H.: Jedd: A BDD-based Relational Extension of Java. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ACM Press (2004)
3. Beyer, D., noack, A., Lewerentz, C.: Efficient Relational Calculation for Software Analysis. IEEE Transactions on Software Engineering (TSE) **31**(2) (2005) 137–149
4. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM **13**(6) (1970) 377–387
5. Palamidessi, C., Stronati, M.: Differential Privacy for Relational Algebra: Improving the Sensitivity Bounds via Constraint Systems. ArXiv e-prints **cs.CR** (2012)
6. Gervet, C.: New structures of symbolic constraint objects: sets and graphs. In: 3rd Workshop on Constraint Logic Programming, Citeseer (1993)
7. Van Cauwelaert, S., Gutiérrez, G., Van Roy, P.: A New approach for Constraint Programming in Music Using Relation Domains. In: ICMC2012, Ann Arbor, MI: MPublishing, University of Michigan Library (2012)
8. Van Cauwelaert, S., Gutiérrez, G., Van Roy, P.: Practical Uses of Constraint Programming in Music using Relation Domains. Journal of the Korean Electro-Acoustic Music Society KEAMSAC2012 (2012)
9. Capeletto, M.: Boost Bimap. Boost Software License v1.0
10. López-Muñoz, J.M.: Boost Multi-index Containers Library. Boost Software License v1.0
11. Beyer, D.: Relational Programming with CrocoPat. In: Proceedings of the 28th ACM/IEEE International Conference on Software Engineering (ICSE 2006, Shanghai, May 20-28), ACM Press, New York (NY) (2006) 807–810
12. Gecode Team: Gecode: Generic Constraint Development Environment. (2006) MIT License
13. Somenzi, F.: CUDD: BDD package, University of Colorado, Boulder. (2013)
14. Flener, P., Pearson, J.: Introducing ESRA, a relational language for modelling combinatorial problems. In: In Proceedings of LOPSTR '03: Revised Selected Papers, Springer (2004) 214–232
15. Hnich, B.: Function variables for constraint programming. AI Communications **16**(2) (2003) 131–132
16. Cadoli, M., Mancini, T.: Combining relational algebra, SQL, constraint modelling, and local search. Theory and Practice of Logic . . . (2007)
17. Le Pape, C., Perron, L., Régin, J.C., Shaw, P.: Robust and Parallel Solving of a Network Design Problem - Springer. Principles and Practice of . . . (2006)
18. Dooms, G., Deville, Y., Dupont, P.: Cp (Graph): Introducing a graph computation domain in constraint programming. In Van Beek, P., ed.: CP 2005: Eleventh International Conference on Principles and Practice of Constraint Programming, Springer (2005) 211–225
19. Dooms, G.: The CP(Graph) Computation Domain in Constraint Programming. PhD thesis, Department of Computing Science and Engineering, Université catholique de Louvain (September 2006)
20. Gervet, C.: Interval propagation to reason about sets: Definition and implementation of a practical language. Constraints **1**(3) (March 1997) 191–244
21. Puget, J.F.: PECOS A High Level Constraint programming Language. In: Proceedings of SPICIS. (1992)
22. Hawkins, P.J., Lagoon, V., Stuckey, P.J.: Solving Set Constraint Satisfaction Problems using ROBDDs. Journal of Artificial Intelligence Research (JAIR) (September 2011) 109–156

23. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: Principles and Practice of Constraint Programming (CP 2007). Lecture Notes in Computer Science, Springer (2007)
24. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. The journal of logic programming **19** (1994) 503–581
25. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge: Cambridge University Press (2002)
26. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell System Technical Journal **38**(4) (1959) 985–999
27. Akers, S.B.: Binary decision diagrams. Computers, IEEE Transactions on **100**(6) (1978) 509–516
28. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. Computers, IEEE Transactions on (8) (1986) 677–691
29. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 1020 States and beyond. Information and Computation **98**(2) (June 1992) 142–170
30. Spiessens, A.: Patterns of Safe Collaboration. PhD thesis, Université catholique de Louvain (February 2007)
31. Spiessens, A., Jaradin, Y., Van Roy, P.: SCOLL and SCOLLAR. Technical Report RR2005-12, Université catholique de Louvain (October 2005)