Data Driven Relational Constraint Programming

Michael Valdron
Ontario Tech University
2000 Simcoe Street North
Oshawa, Ontario L1G 0C5
Email: michael.valdron@ontariotechu.ca

Ken Q. Pu
Ontario Tech University
2000 Simcoe Street North
Oshawa, Ontario L1G 0C5
Email: ken.pu@ontariotechu.ca

***Abstract*—We propose a data-driven constraint programming environment that merges the power of two separate domains: databases and SAT-solvers. While a database system offers flexible data models and query languages, SAT solvers offer the ability to satisfy logical constraints and optimization objectives. In this paper, we describe a goal-oriented declarative algebra that seamlessly integrates both worlds. Bring from proven practices in functional programming, we express constants, variables and constraints in a unified relational query language. The language is implemented on top of industrial strength database engines and SAT solvers.**

In order to support iterative constraint programming with debugging, we propose several debugging operators to assist with interactive constraint solving.

I. Introduction

Constraint satisfaction problems have a long history. There have been numerous breakthroughs [1] [2] that resulted in modern SAT solvers that can solve substantial problems at the industrial scale.

Concurrent with the development of constraint satisfaction solvers, we have seen tremendous advances in the implementations of large scale database systems. With the growing reliance on data-driven decision making both by software and users, databases are playing a central role in systems and organizations. Thus, safety and data quality have become significant issues associated with data-driven decision making. While database systems often offer some level of integrity checking, these features mostly deal with data integrity verification (typically done in polynomial time), but not tasks such as constraint satisfaction, or optimization (typically are NP-complete). These tasks are perfectly suitable for SAT-solvers. Unfortunately, there have been little efforts in incorporating SAT solvers into the data processing pipeline.

We are motivated to build new data processing solutions based on elements from the two pillars: databases and constraint satisfaction. In this paper, we present a well-defined abstraction in the form of a data-driven constraint programming language. The language has a core algebra, which is based on the nested relational algebra of SQL but also extends with constraint-based operators. The language allows users to compose the algebraic operators to transform data into a set of constraints consisting of both data and variables. At the implementation level, this is done using a combination of database queries and data transformation. Next, one can invoke the SAT solving operators to generate the solution if the constraints are feasible. This is done using a constraint solver. The solution is then stored back to the database.

From modern software engineering practices, we have learned that the most significant challenge in software development is to mitigate software bugs based on human error. To this end, programming languages are equipped with type systems and interactive read-evaluate-print-loop (REPL) interfaces. It has been shown that a static type system can be extended to constraint programming [3]. In this paper, we will further extend the experience of constraint programming by operators for interactive constraint satisfaction. The equivalence to a program runtime failure is if the constraints are infeasible, i.e. no solutions exist. Our implementation offers conflict detection and conflict repair operators, which can be used to identify sets of conflicting constraints, and automatically generate feasible subproblems by disabling a minimal number of constraints. These features are designed to support the user in understanding the conflicts that exist in the constraint program.

We introduce two debugging operators: conflict detection and minimal repairs. We demonstrate that they are efficiently computable sets, and these sets can help the user to identify the source of infeasibility within constraint programs, as well as automatic repairs.

The organization of the paper is as follows:

- Section II formally defines the unified data model for nested relational constraint satisfaction problems. We define an algebra with both data manipulation and constraint construction operators. Coupled with the standard functional programming instructions, one can use them to express complex constraint programs in various domains.
- Section III defines several operators to support iterative constraint programming development. These operators are designed to be invoked interactively to identify conflicts in infeasible programs, or minimal repairs of infeasible programs. We will also describe how our framework permits intuitive and informative visualization of such conflicts or repairs.
- Section IV provides the implementation and evaluation details.
- Section V surveys related work, and highlights the major differences of our work compared to previous literature.
- Finally, Section VI summaries our findings and discusses future directions of this research program.

II. Constraint relational algebra

In this section, we will define a model for constraint relations. The model generalizes the first-order relational model and its query language of relational algebra with aggregation [4]. The relational model is extended with two fundamentally new data types: variables and constraints. We also extend the relational algebra with aggregation with arithmetic and aggregation operators. These extensions are to handle variables and first-order constraints.

For readability, we choose to express the foundation of relational bases (data model and the relational algebraic operators) in terms of the well-established syntax of Structured Query Languages (SQL). We believe that this makes our extensions more intuitive and readable without loss of rigour.

A. Relational model with constraints

The current relational model adopted by modern database systems support relations with strongly-typed columns. The SQL standard defines the types of columns. Typical column types are: FLOAT, TEXT, VARCHAR, TIMESTAMP, and more. Databases such as PostgreSQL also supports user-defined ENUM and user-defined types. So, it is possible to create types such as CITY, or DAY_OF_WEEK. We will denote the data types as τ_{data}.

We extend the relational model with two fundamentally new column types: variables and constraints. A relation r can have columns $r(c_1, c_2, \dots c_n)$. Each column c_i has a type τ where τ can be one of the data types as permitted by the SQL standard, or:

- VAR[τ_{data}]
- Constraint

For example, we can have the following table schema in the extended constraint relational model.

```
CREATE TABLE T (
    a INTEGER,
    b INTEGER,
    x? VAR[INTEGER]
)
```

Instances of such schema would contain two columns (a and b) of integers, but the column $x?$ contains variables whose values are unspecified, and could only be determined by solving constraints. Variable constructors can create variable instances.

We can insert into an instance of T as:

```
INSERT INTO T(a, b, x?)
VALUES (10, 20, new_var())
```

Since we know that $T.x$ has the type Var[Integer], the constructor `new_var` does not need type hints.

Next we need *constraint* columns. A column can have the type Constraint, and its instance will be a logical formula defined by variables, data constants and constraint operators. To separate constraint operators from SQL logical operators, we use the notation $:<:$ instead of $<$. The former is a constraint operator, while the latter is the regular SQL comparison operator. Similarly, all constraint operators will be written in the form of $:\circ:$.

For example, here is a table with a constraint column and a text column. Note, the variable used in this constraint is unbounded.

```
CREATE TABLE U(c CONSTRAINT, comment TEXT);
INSERT INTO U
  VALUES ((0 :<: x? :and: x? :<: 10),
          'small numbers');
```

Typically, we work with constraints whose variables are bounded. Referring to the table T above, we can now define a new table U based on T with a constraint column which specifies that x is in between $[a, b]$, expressed as $(a :<: x)$ `:and:` $(x :<: b)$.

```
CREATE VIEW V(c CONSTRAINT) AS
 (SELECT (a :<: x? :and: x? :<: b)
  FROM T)
```

The above example demonstrates a powerful aspect of relational algebra, namely deriving new columns in views. With our constraint relational model, one can derive new columns using the following mechanisms. The view `V` as defined above uses expression operators `a :<: x?`, and constraint operators `... :and: ...`.

- new variables using the `new_var` constructor
- Expression operators such as `:+:`, `:-:`, `:*:`, `:/:`
- new constraints using constraint operators.

Other constraint operators are: `:not:`, `:or:`, `:imply:`, `:different:`, and more.

B. Aggregation queries of constraint relations

We also extend relational algebra with aggregation in a natural way. Aggregation can be performed on variables and expressions by compatible aggregation functions with the data type of the variables.

For example, suppose we have a relation:

```
CREATE TABLE R (
    item_id INTEGER PRIMARY KEY
    color ENUM('red', 'green', 'blue')
    x? VAR[Integer]
    y? VAR[enum('red', 'green', 'blue')]
) AS ...
```

It has two variable columns with data types of `Integer` and `enum(...)`. We can apply numerical aggregation (e.g. `:sum:`, `:max:`, `:min:`) to `x?` because its' underlying data type is numerical. The result is not a number, but rather, it will be an expression involving one or more variables. Since `y?` variable in R has the data type of `enum`, we can only use a restricted set of aggregation functions: `:count:` and `:countdistinct:`.

The following query generates a relation with two columns (color, total?). Note that `total?` is a variable column of type Var[Integer].

```
SELECT color, :sum:(x?) as total?
FROM R
GROUP BY color
```

The following query generates variables for each distinct color in R that represents the distinct number of solutions of y?.

```
SELECT color, :count:(y?) as c?
FROM R
GROUP BY color
```

Finally, we extend aggregation to include constraint columns. We support two types of aggregation: conjunctive aggregation :every: and disjunctive aggregation :some: . The semantics is that :every: $(x?)$ constructs a new boolean variable which is true if and only if every instance of $x?$ is true in the respective group-by partition. Similarly, :some: $(x?)$ is a boolean variable that is true if and only if at least one $x?$ is true in the group-by partition.

For the relation R(color, x?, y?) defined above, we can apply the following group-by query using aggregation of constraints.

```
SELECT color,
       :every:(x? :>: 10) AS c1?,
       :some(y? :=: 'blue') AS c2?
FROM R
GROUP BY color
```

C. Solving constraint relations

Section II-A and Section II-B describes how we can transform relations with constraints and variables using the extended query language. The eventual goal is to find solutions of *variables* so that *goals* are satisfied. This eventual goal is fundamentally new in constraint programming compared to database query processing: we need to declare *goals* explicitly to be satisfied. In this section, we define the semantics and a simple syntactic extension to SQL for goal declaration.

Global variable scope: We assume that all variables that are stored in *tables* and *views* are indexed in a global registry. Some variable object identifier can uniquely identify each variable. The global variable scope is a collection of variable objects:

$$V = \{v_1, v_2, \ldots, v_n\}$$

The variable scope V is updated by base constraint relations as well as creations of views that have derived expression and variable constructors. Therefore, executing a query like CREATE VIEW ... may trigger new variables being added to the global variable scope V.

Global goal scope: We assume a goal is a complex constraint. Each goal is identified by a key which can be a unique user-defined name assigned to the goal, or a tuple of data values. Thus, a goal can be modeled as a key-constraint pair: $k \mapsto \theta(x_1, x_2, \ldots, x_n)$ where k is the goal's key and θ the constraint. The variables of a goal must be defined in the global variable scope. The global goal scope is a mapping of keys to constraints:

$$G = \{k_i \mapsto \theta_i : i = 1 \ldots m\}$$

To add goals to G, we need to do so with a G-mutation operator explicitly:

CREATE GOAL *[name]* AS θ

where the key for the goal is *name*, and the constraint is θ. If *name* is omitted, the system will assign a unique randomized string as the goals key. Note that θ can be constructed as the result of an aggregation query over a constraint relation.

To add multiple goals to G, we use:

```
CREATE GOALS AS
    SELECT $a_1, a_2, a_i, \theta$
    FROM ...
```

Where $(a_1, a_2, \ldots a_i)$ are data expressions, and θ is a constraint expression. This will add multiple goals to G, with each role in the SELECT query being one goal. Each goal has its key as $(a_1, a_2, \ldots a_i)$, and the constraint as θ.

Using CREATE GOAL and CREATE GOALS form G. Thus, the final constraint SAT program is $\langle G, V\rangle$.

We trigger the solver to find solutions to the variables when we select from the SOLUTION(r) where r is a relation or view containing variables.

```
SELECT $x?, y?, \ldots$
FROM SOLUTION( $r$ )
  ...
```

Note that we are selecting from not the relation r, but its solution SOLUTION(r). The selected variable columns $x?$ and $y?$ will be replaced with data values from their solutions. Expression columns and constraint columns will also be data values.

In Section III, we will look at additional solution triggering operators to support iterative and interactive constraint debugging.

D. A case study

Suppose we have a database of cities.

```
Cities
| city              | state |
|-------------------|-------|
| San Francisco     | CA    |
| Los Angeles       | CA    |
| Las Vegas         | NV    |
| Salt Lake City    | UT    |
| ...               | ...   |
```

We also have information on the type of tourist activities in each city.

```
CityActivities
| city              | activity   |
|-------------------|------------|
| San Francisco     | Shopping   |
| San Francisco     | Hiking     |
| Los Angeles       | Shopping   |
| Los Angeles       | Restaurant |
| Las Vegas         | Shopping   |
| Las Vegas         | Theatre    |
| Salt Lake City    | Hiking     |
| ...               | ...        |
```

We want to plan a road trip with the following constraints:

C1: The trip lasts 3 days. We visit a different city each day.
C2: We wish to visit at least two states.
C3: We will do two activities each day (morning and afternoon).
C4: We want to experience at least 5 different experiences.

We declare the following variables.

```
TravelPlan
| day | city? |
|-----|-------|
|  1  |  ?    |
|  2  |  ?    |
|  3  |  ?    |
```

We also need the activity variables.

```
ActivityPlan
| day | time      | act? |
|-----|-----------|------|
|  1  | morning   | ?    |
|  1  | afternoon | ?    |
|  2  | morning   | ?    |
|  2  | afternoon | ?    |
|  3  | morning   | ?    |
|  3  | afternoon | ?    |
```

The next step to impose the goals:
C1: all cities are distinct

```
CREATE GOAL AS
  SELECT :distinct:(city?)
  FROM TravelPlan
```

C2: visit two different states
The first step is that we create new variables that represent the states to be visited.

```
CREATE VIEW TravelPlanStates AS
 SELECT city?, new-var() as state?
 FROM TravelPlan
```

This query produces a relation of variables over both cities and states.

```
TravelPlanStates
| day | city? | state? |
|-----|-------|--------|
|  1  |  ?    |   ?    |
|  2  |  ?    |   ?    |
|  3  |  ?    |   ?    |
```

First, we must restrict the states variable `state?` to be the state of the city variable `city?`. To do this, we join `TravelPlanStates` with `Cities` in order to compute multiple constraints, each will be set as a goal.

```
CREATE GOALS AS
  SELECT
    name, state,
    (city? :=: name :=>: state? :=: state) as c
  FROM TravelPlanStates
  JOIN Cities
```

Now, we can express the state coverage constraints.

```
CREATE GOALS AS
  (SELECT :countdistinct:(state?)
   FROM TravelPlanStates)
   :>=: 2)
```

Next, we express the constraint of two activities each day. First, we change the layout:

```
CREATE VIEW ActivityPlan2 AS
  SELECT day,
         S.act? AS act_morng?,
         T.act? AS act_aftn?
  FROM ActivityPlan S
  JOIN ActivityPlan T
  USING (day)
  WHERE S.time = "morning"
    AND T.time = "afternoon"
```

The view has the same set of activity variables but arranged into two separate columns.

```
ActivityPlan2
day | act_morng? | act_aftn? |
----|------------|-----------|
 1  |      ?     |     ?     |
 2  |      ?     |     ?     |
 3  |      ?     |     ?     |
```

We can now assert that the morning and afternoon activity variables must be distinct.

```
CREATE GOALS
  SELECT act_morng? <> act_aftn?
  FROM ActivityPlan2
```

Now, we need to restrict the activity variables `act?` to be available in the city `city?` according to the `CityActivities`. One of these constraints looks like: *if* `city?` = *"Las Vegas"— then (*`act?` = *"Shopping" or* `act?` = *"Theatre")*

This constraint can be expressed in first-order logic as:

$$\begin{aligned} & \texttt{city?} = \text{LasVegas} \\ \Longrightarrow\; & \texttt{act?} = \text{Shopping} \vee \texttt{act?} = \text{Theatre} \end{aligned}$$

Equivalently, we can also express it in aggregated form:

$$\sum \begin{array}{l} \texttt{city?} = \text{LasVegas} \implies \texttt{act?} = \text{Shopping} \\ \texttt{city?} = \text{LasVegas} \implies \texttt{act?} = \text{Theatre} \end{array}$$

To impose all such constraints, we make use of constraint aggregation in the query language.

```
CREATE GOALS
 SELECT
  city,
  :some:(city? :=: city :=>: act? :=: act)
  FROM ActivityPlan
  JOIN CityActivities
  GROUP BY  city?, city
```

The final constraint is that we cover at least five different activities. Counting the distinct activities achieves this.

```
CREATE GOAL
  SELECT :countdistinct:(act?) :>=: 5
  FROM ActivityPlan
```

III. Iterative Constraint Programming

Given the data-driven nature of the relational constraint programming, we can potentially generate thousands of goals over millions of variables. It becomes exceedingly easy for some of the constraints to be conflicting, and therefore the whole relational constraint program $\langle G, V\rangle$ (as defined in Section II-C) to be infeasible.

The infeasibility of $\langle G, V\rangle$ might be due to several reasons:

- Error in the relational constraint queries
- Inherit conflicts in the data
- Unreasonable expectations of the goals

We must provide support by providing the developer tools to identify the source of infeasibility. We want to provide the following debugging capabilities:

- Localize the conflicting constraints
- Provide automatic constraint repair in order to generate partial solutions

A. Conflict Detection

Definition 1 (Conflict Sets): A set $S \subset G$ of constraints is a conflict set if $\langle S, V\rangle$ is infeasible. S is a min-conflict set if S is a conflict set, but none of its subsets are conflict sets.

Conflicts can occur in complex ways. When the entire goal scope $\langle G, V\rangle$ is infeasible, we want to compute some min-conflict set S so that we can examine the source of the infeasibility.

Conflict detection operator accomplishes this:

$$\mathrm{con} : G \mapsto S$$

where $\mathrm{con}(G) \subseteq G$ is just one of the conflict sets.

We can compute $\mathrm{con}(G)$ in two different ways: bottom-up or topdown. In the bottom-up approach, we start with $S = \emptyset$, and incrementally add goals until we find a S that is infeasible.

```
Algorithm: con with bottom-up computation
let $\mathcal{X} = \mathrm{powerset}(G)$
let $\mathcal{X}$ = sort $\mathcal{X}$ by cardinality
for $X$ in $\mathcal{X}$
    if $\langle X, V\rangle$ is infeasible:
        return $X$
    end if
end for
```

Another approach is to compute $\mathrm{con}(G)$ using a topdown approach. We start with G and remove elements until $G - X$ becomes feasible. We return the previous infeasible goal set as $\mathrm{con}(G)$.

```
Algorithm: con with topdown computation
let $X = S = G$
for $g \in G$
    let $X' = X - \{g\}$
    if $X'$ is feasible:
        return $X$
    else
        let $X = X'$
    end if
end for
```

B. Optimal Conflict Repair

The $\mathrm{con}(G)$ operator returns one of the conflict sets in G to help users understand the source of conflict. Unfortunately, this may still result in a large number of mutually conflicting goals. We want to be able to automatically disable the minimal number of goals so that the problem becomes feasible.

Definition 2 (Repair Sets): Let $R \subseteq G$ be a set of goals. It is a *repair* set if $\langle G - R, V\rangle$ is feasible. R is a *min-repair* set if no subset of R is a repair set.

We introduce a repair operator:

$$\mathrm{repair} : G \mapsto R$$

It turns out that we can pose the *optimal* repair as a MAX-SAT optimization problem where we introduce a linear maximization object to the SAT problem:

$$\langle G, V, h\rangle$$

Where h is a score function to be maximized. MAX-SAT allows one to find solutions of V that satisfies all the goals in G *and* while maximizing the function h.

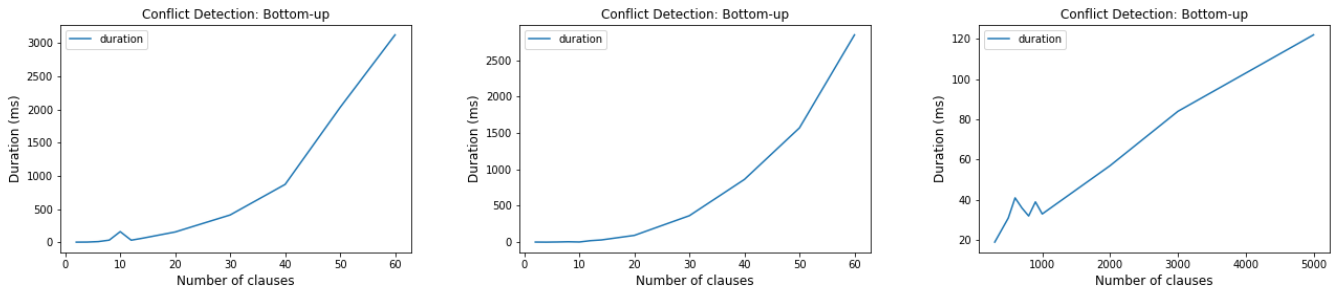
We can make use of MAX-SAT to solve the min-repair problem. For each goal $G_i = \theta_i$ we define a new enable boolean variable α_i that can be used to *disable* the goal G_i. This is done by replacing G_i with a new goal $G'_i = (\alpha_i \implies \theta_i)$. If $\alpha_i = 0$, then G_i does not need to be satisfied since $0 \implies 0$ is still true.

So, we define a new variable scope as $V' = V \cup \{\alpha_i : i \in G\}$, and a new goal scope as $G' = \{\alpha_i \implies \theta_i : i \in G\}$. Now, we can define the maximization function $h = \sum_i \alpha_i$. Namely, we try to enable as many goals as possible and still remain feasible.

```
Algorithm **repair** using MAXSAT
let $V' = \{\alpha_i \models \mathrm{boolean} : i \in \mathrm{keys}(G)\}$
let $G' = \{i \mapsto (\alpha_i \implies \theta_i) : i \in \mathrm{keys}(G)\}$
let $h = \sum_{i \in \mathrm{keys}(G)} \alpha_i$
MAXSAT$(G', V', h)$
return $\{i : \mathrm{solution}(\alpha_i) = 0\}$
```

IV. Implementation and Evaluation

The entire system is implemented on top of Google OR-Tools [5] and custom application code in using the Clojure



(a) Conflict Set Detection (bottom-up) (b) Conflict Set Detection (topdown) (c) Optimal Repairs

Fig. 1: Performance of interactive debugging operators

programming language. We have implemented the core constraint relational algebra and utilizes the Google OR-Tools to perform the SAT solver and the MAX-SAT solver.

In this section, we present some experimental evaluation of our implementation.

A. 3-CNF

Following the standard evaluation of SAT solvers [6], [7], we generate random 3-CNF instances consisting of a set of randomized disjunctive clauses containing literals of variables or their negations.

Figure 1 shows the performance characteristics of the debugging operators. We observe that both implementations of con are more costly than the repair operator. This observation suggests that as an interactive tool, the system can incrementally make auto-suggestions on how constraints should be disabled to make the while problem feasible.

B. Segment Overlapping

For the second round of experiments, we use the problem we call the *Segment Overlapping* problem. The key here is to use the *Segment Overlapping* problem to provide an additional evaluation of the performance of the algorithms defined back in Section III to further show the value in using data-driven *goals*. To begin, we must define a few concepts.

Definition 3: A **Segment** is an abstract collection of variables which includes a lower bound (segment start point) S_{lower}, a segment length S_{length}, and a upper bound (segment end point) S_{upper}.

Each segment also include a hidden constraint which is defined as $S_{\text{length}} = S_{\text{upper}} - S_{\text{lower}}$. This constraint ensures that the S_{length} does not break the restrictions imposed by the bounds S_{lower} and S_{upper}.

Definition 4: A **NoOverlap** is a constraint that enforces that all segments associated with it are unable to overlap each other. For more a formal definition of this constraint see [8].

In our problem, we only consider segment pairs for the **NoOverlap** constraints, which we abstract to *goals*. The *Segment Overlapping* problem is set up the following way:

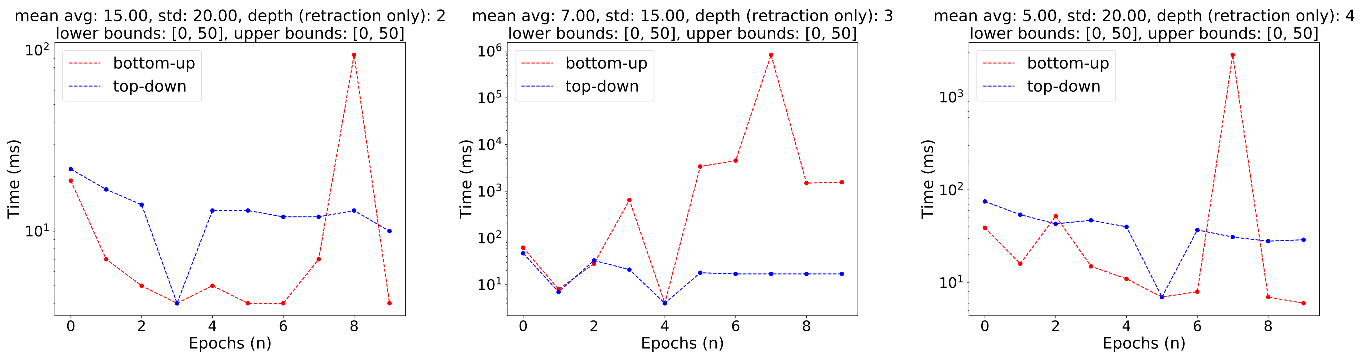
- There exists predefined number of segments S_n, all with a $S_{\text{length}} \sim \mathcal{N}(\mu, \sigma^2)$ which is randomly sampled, a $S_{\text{lower}} \in \mathcal{D}_{\text{lower}}$, and a $S_{\text{upper}} \in \mathcal{D}_{\text{upper}}$
- There are **NoOverlap** constraints for each of the segment pairs, such that there is $\binom{n}{2}$ constraints
- The overall problem is created from parameters and a Database

The problem is then run against the debug algorithms over the number of epochs n, which recreates the problem with different sampled S_{length} for every segment. We ran this overall experiment in three stages with the same n but with altered parameters. A parameter *depth* is used as an additional parameter for the topdown version of con denoted as $\mathrm{con_t}$ to specify how many conflict-sets to pass over before returning one. *depth* is added for better efficiency of $\mathrm{con_t}$. The rest of the changes in the experiments are done with the Databases. There are three Databases for each experiment, and each has input data that differ between experiments. The attributes of these Databases are as follows:

- The number of segments S_n
- The mean μ of the normal distribution of S_{length}
- The standard deviation σ of the normal distribution of S_{length}
- The lower bound S_{lower} (left the same in all experiments)
- The upper bound S_{upper} (left the same in all experiments)

The alterations done in each of the three can be seen in Figures 2a, 2b, and 2c as well as the performance outcomes. All of the Figures in Fig. 2 shows the amount of runtime in milliseconds over each epoch. The Figures y-axis are in log scale to both properly show the smaller and larger changes in the runtimes.

Figure 2a shows the experiment with the smallest amount of segments and constraints. It has a large μ and σ. The *depth* parameter is only 2 as it will likely not take long to find conflicts in a pool of 10 constraints. Figure 2b shows the experiment with the in-between number of segments and constraints. Unlike the other experiments, this one has a smaller σ, and this has a significant aspect that we will get



(a) 5 Segments, 10 Constraints (b) 8 Segments, 28 Constraints (c) 10 Segments, 45 Constraints

Fig. 2: Segment Overlapping

back to later. The μ is also smaller, with the increase of the segments within the same bounds domains $\mathcal{D}_{\text{lower}}$ and $\mathcal{D}_{\text{upper}}$. Figure 2c shows the experiment with the largest problem of 10 segments and 45 constraints. This experiment, however, has the same σ as the first but has the lowest μ of them all.

Notice from the figures where conflicts occur, the points close to zero milliseconds are runs, which immediately return nothing as the whole problem is *feasible*. The spike points in the figures, on the other hand, are points when the problem was *infeasible*, and the algorithms had to run a search for conflict.

Let us return to the smallest σ experiment shown in Figure 2b, this one had the more significant spike points. This pattern is due to the lower σ this experiment has. The pattern shows that the segment lengths are more likely to hover around the middle, causing fewer conflicts to happen with fewer constraints or with lots of constraints, which causes conflict searching to become longer for specific heuristics. The heuristic of the issue with this as the figures show is bottom-up $\mathrm{con_b}$, this unlike the $\mathrm{con_t}$ heuristic does not hit conflicts right away or has the efficiency benefit of using the *depth* parameter as a stopping point. All this, of course, also depends on the μ being closer to the middle of the chosen bounds.

To look at pattern further we have compiled the averages and standard deviations of the huerstics' runtime results. The similar as the figures, these results will be presented in scientific notation. For $\mathrm{con_b}$, we have an average μ_b of 2.82×10^4 milliseconds and a standard deviation σ_b of 1.52×10^5 milliseconds. For $\mathrm{con_t}$, we have an average μ_t of 2.40×10^1 milliseconds and a standard deviation σ_t of 1.67×10^1 milliseconds.

We can immediately see that μ_b outruns the other average times μ_{rand} and μ_t as this pattern is seen in the figures. If we look at σ_b, the value is also high, meaning that $\mathrm{con_b}$ has significant runtime differences throughout epochs, which also the pattern can be seen in the figures. From seeing this and the figures, we can see that $\mathrm{con_b}$ runtimes depend on the placement of the conflicts within the powerset.

Why does all this matter in showing the value of data-driven *goals*? All this shows that we can efficiently perform these kinds of analyses with data to drive the problem.

V. Related Work

Constraint Programming is useful for one to use for any problems which are NP-Complete, discrete, and deterministic [9]. SAT Solvers are one of the solutions to implementing constraint programming models. OR-Tools SAT Solver, an open-sourced solver developed by Google [5], is one of the most modern efficient SAT solvers to date. OR-Tools has won first, second, and third place in its ability to solve many of the problems in the recent MiniZinc challenges [7][6], parallelized challenges in particular. It is due to OR-Tools' efficient and modernized implementation, which is why we chose to target it for our work. Other well known SAT Solvers include *Chaff* [10], *MINISAT* [11], *Tinisat* [12], *FznTini* [13], and *BEE* [14]. More information on SAT Solvers can be found in [15][16].

Constraint Programming is not just limited to SAT Solvers, *Prolog Systems*, are a form of CP solvers that uses *Prolog*, a logic programming language [17], also allow one to design a CP model programmatically to solve a Boolean Satisfiability Problem (CSP). One such *Prolog System* used in most of the related works is the GNU Prolog Solver [18] implemented as an open-sourced solution to do constraint programming in *Prolog*.

Debugging *Constraint Programming* models provide one with a better understanding of what is happening with the variables and constraints during evaluation. This is why research into conflict detection, model data abstraction, and annotating items in CP Models has been studied in plenty of other works. What sets our work from most other forms of solutions to the problem presented in our paper is that these solutions are typically designed to be incorporated into CP/SAT Solver implementations, such as the conflict-driven learning strategy purposed in [19] which is used within Google's OR-Tools SAT Solver [5] implementation we target. More information on what is included in modern SAT Solvers can be found in [20].

Other works similar to ours look into a method of abstraction called S-Boxes for Visualizing CP debugging [21] [22].

This work also goes into the idea of *Goals*, which in this work is defined as being S-Boxes within S-Boxes [21]. The last conflict policy is another strategy to find relevant conflicts, and a method that follows this is proposed in [23]. As implied, the last conflict policy says one is to assume the last culprit found in the search tree during the propagation phase is the culprit, most likely causing grief in the *Constaint Network* of a CP model [23].

Data abstraction of CSPs is not just excellent for conflict detection, but also for merely visualizing the problems those needing to see what is happening. An example of this includes the work [21], which was mentioned earlier. Another solution done by [24] purposes storing variables and constraints in the CSP within an XML data structure with annotations to properly abstract the data to users using their CP visualization tools.

It is due to the significance of and reasons for needing to abstract CSPs into high-level pieces of information, which has motivated us to bring a solution similar to these to the modern interactive programming environment for a better experience tackling CSPs.

VI. Conclusion and Future Work

We have presented a relational constraint programming framework that allows data-driven constraint satisfaction solving. Our design merges the power of relational algebra with constraint solving. We have extended the SQL language to include computation and aggregation of variables and constraints, which are used to create goals.

To support interactive and iterative debugging, we introduced two operators: con and repair that computes the conflicting set and repair set, respectively. We have shown that they can be used at scale to help the system to assist with identifying issues and repairs of the constraint program.

Future work includes the static analysis and query optimization of the constraint program. We are also interested in generating visualizations of intermediate and final results.

References

[1] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI magazine*, vol. 13, no. 1, pp. 32–32, 1992.

[2] P. Codognet and D. Diaz, "Compiling constraints in clp(FD)," *The Journal of Logic Programming*, vol. 27, no. 3, pp. 185–226, Jun. 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0743106695001212

[3] F. Fages and E. Coquery, "Typing constraint logic programs," *Theory and practice of logic programming*, vol. 1, no. 6, pp. 751–777, 2001.

[4] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases*. Addison-Wesley Reading, 1995, vol. 8.

[5] L. Perron and V. Furnon, "Or-tools," Google. [Online]. Available: https://developers.google.com/optimization/

[6] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, "The MiniZinc Challenge 2008–2013," *AI Magazine*, vol. 35, no. 2, pp. 55–60, Jun. 2014, number: 2. [Online]. Available: https://aaai.org/ojs/index.php/aimagazine/article/view/2539

[7] P. J. Stuckey, R. Becket, and J. Fischer, "Philosophy of the MiniZinc challenge," *Constraints*, vol. 15, no. 3, pp. 307–316, Jul. 2010. [Online]. Available: https://doi.org/10.1007/s10601-010-9093-0

[8] P. Hentenryck, V. Saraswat, and Y. Deville, "Design, implementation, and evaluation of the constraint language cc(FD)," in *Constraint Programming: Basics and Trends*, G. Goos, J. Hartmanis, J. Leeuwen, and A. Podelski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, vol. 910, pp. 293–316, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-59155-9_15

[9] E. E. Ogheneovo, "Revisiting Cook-Levin theorem using NP-Completeness and Circuit-SAT," *International Journal of Advanced Engineering Research and Science*, vol. 7, no. 3, Mar. 2020, number: 3. [Online]. Available: http://journal-repository.com/index.php/ijaers/article/view/1749

[10] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver," in *Proceedings of the 38th annual Design Automation Conference*, ser. DAC '01. Las Vegas, Nevada, USA: Association for Computing Machinery, Jun. 2001, pp. 530–535. [Online]. Available: https://doi.org/10.1145/378239.379017

[11] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds. Berlin, Heidelberg: Springer, 2004, pp. 502–518.

[12] J. Huang, "A Case for Simple SAT Solvers," in *Principles and Practice of Constraint Programming – CP 2007*, ser. Lecture Notes in Computer Science, C. Bessière, Ed. Berlin, Heidelberg: Springer, 2007, pp. 839–846.

[13] ——, "Universal Booleanization of Constraint Models," in *Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, P. J. Stuckey, Ed. Berlin, Heidelberg: Springer, 2008, pp. 144–158.

[14] A. Metodi, M. Codish, and P. J. Stuckey, "Boolean Equi-propagation for Concise and Efficient SAT Encodings of Combinatorial Problems," *Journal of Artificial Intelligence Research*, vol. 46, pp. 303–341, Mar. 2013, arXiv: 1402.0568. [Online]. Available: http://arxiv.org/abs/1402.0568

[15] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, Apr. 2005. [Online]. Available: https://doi.org/10.1007/s10009-004-0183-4

[16] S. Alouneh, S. Abed, M. H. Al Shayeji, and R. Mesleh, "A comprehensive study and analysis on SAT-solvers: advances, usages and achievements," *Artificial Intelligence Review*, vol. 52, no. 4, pp. 2575–2601, Dec. 2019. [Online]. Available: https://doi.org/10.1007/s10462-018-9628-0

[17] W. Clocksin and C. S. Mellish, *Programming in Prolog: Using the ISO Standard*, 5th ed. Berlin Heidelberg: Springer-Verlag, 2003. [Online]. Available: http://www.springer.com/gp/book/9783540006787

[18] D. Diaz and P. Codognet, "Design and Implementation of the GNU Prolog System," *Journal of Functional and Logic Programming*, vol. 2001, no. 6, 2001.

[19] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, Nov. 2001, pp. 279–285, iSSN: 1092-3152.

[20] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva, "Empirical Study of the Anatomy of Modern Sat Solvers," in *Theory and Applications of Satisfiability Testing - SAT 2011*, ser. Lecture Notes in Computer Science, K. A. Sakallah and L. Simon, Eds. Berlin, Heidelberg: Springer, 2011, pp. 343–356.

[21] F. Goualard and F. Benhamou, "A visualization tool for constraint program debugging," in *14th IEEE International Conference on Automated Software Engineering*, Oct. 1999, pp. 110–117.

[22] ——, "Debugging Constraint Programs by Store Inspection," in *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, ser. Lecture Notes in Computer Science, P. Deransart, M. V. Hermenegildo, and J. Małuszynski, Eds. Berlin, Heidelberg: Springer, 2000, pp. 273–297. [Online]. Available: https://doi.org/10.1007/10722311_12

[23] C. Lecoutre, L. Saïs, S. Tabary, and V. Vidal, "Reasoning from last conflict(s) in constraint programming," *Artificial Intelligence*, vol. 173, no. 18, pp. 1592–1614, Dec. 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370209001040

[24] F. Fages, S. Soliman, and R. Coolen, "CLPGUI: A Generic Graphical User Interface for Constraint Logic Programming," *Constraints*, vol. 9, no. 4, pp. 241–262, Oct. 2004. [Online]. Available: https://doi.org/10.1023/B:CONS.0000049203.53383.c1