Platform Agnostic Constraint Modeling using XML

Sumit Haswar
Computer Engineering and Computer Science
California State University Long Beach
Long Beach, USA
sumit.haswar@gmail.com

Dr. Todd Ebert
Computer Engineering and Computer Science
California State University Long Beach
Long Beach, USA
todd.ebert@csulb.edu

***Abstract*— Constraint Programming is a paradigm in Computer Science through which associations between entities are expressed in the form of one or more constraints. It has a stark contrast from traditional imperative programming in the sense that rather than listing steps to find the solution to a problem it lists characteristics of the solution itself. Constraint Modeling is the process of formulating a correct specification to represent those entities in the form of variables and the constraints that dictate them. This paper proposes a methodology for constraint modeling using XML or JSON which are widely used language for data and markup specification, they are both human and machine readable. The model specifications can then be mapped to internal models and used to determine solution to the underlying problem. Some of the important aspects addressed in this paper are: modeling of variables and domains, subsets, local and global constraints and objective functions. Modeling of solution using XML is also described. A special section is dedicated for representation of linear programming as they are widely encountered in the world of constraint processing.**

Keywords—constraint programming; constraint modeling, constraint modeling language; solver independent constraint specification; XML modeling

I. INTRODUCTION

The process of constraint modeling is defining a set of decision variables, and a set of constraints on those variables that a solution must satisfy. There are two stages in solving a Constraint Programming (CP) problems. State one is the modeling aspect that is breaking up the real world problem and entities into variables, domain and constraints and stage two is using the solver to find the appropriate solution. This document contains an introduction to and specification of modeling a constraints programming problem using EXtensible Markup Language (XML) and JavaScript Object Notation (JSON). The goal of this document is to achieve a universal modeling format using XML and JSON for specifying constraints, variables, functions, constants and other entities that are components of a constraint programming problem. Through this format it should be possible that some program/API will be capable of encoding Model of the problem in a universal format and another program/API should be able to decode it. No other information will be shared by the entities other than the model of the problem in the agreed format.

We are targeting modeling using both XML and JSON format which emphasize on simplicity, generality and usability. This feature can be leveraged to achieve utilization of distributed network of solvers which will communicate with each other to find solution to a constraint problem. Since XML and JSON are popular data-formats and are used by all major technologies. Most programming languages have support of native APIs to parse, generate and query data in either of these formats. Since XML and JSON are all text, use of symbols for variables, keywords and models although is not prohibited it is sort of not encouraged. Hence the format uses only text keywords to represents functions, variables, directives etc.

Through this paper we have tried to lay down the foundation for development of a specification which is platform/language agnostic of the actual Constraint Solver.

The eco-system of a modern CP solving platform can be roughly divided into three paradigms: the modeling or specification aspect, the actual solver which provides the solution and the translator which acts as a mapper between solver and consumer which formulates a CP model using XML or JSON specification. In an ideal scenario the translator or mapping module should be an internal part of the solver itself.

II. MODEL DEFINITION

A. Model Definition using XML

Model definition of a constraint optimization or satisfaction problem encompasses all information regarding the problem. Its variables, constraints, indexers if any. It enables one to define the model for some problem. . In XML modeling the Model tag (<model></model>) acts as the root node of our XML or JSON document. It wraps around all the information for the underlying constraint problem definition. The model element can have a name property which can be any user defined identifier of the model. All other information are defined inside the model tag grouped under relevant elements. The anatomy of a model definition in XML consists of the following key components

```
<model name="puzzle">
    <variables>
      <var name="a" type="bool" />
      <var name="b" type="bool"/>
      <!--other variables-->
    </variables>

    <constant>
      <!--model constants-->
    </constant>

    <constraints>
      <!--model constraints-->
    </constraints>
</model>
```

A lot can be inferred from the above XML snippet, as said earlier model tag forms the root of the specification everything else related to the model is part of the root tag. Variables which belong to the model are grouped under the variables tag, which contains the listing of all variables along with their relevant information like name, type, domain etc. Constants if any are listed inside the constant tag, they should have a name and a value property. All constraints for the model are listed under the constraint tag one below the other.

B. Constants and Literal Definition using XML

Constants and literals are immutable values whose values are directly mentioned in the model specification. Literals only have a value aspect and do not have a name or any other property and its datatype is inferred from the context. Constants need to have a name property so that they can be referenced elsewhere in the constraint model. It is assumed that a constant has been defined before its first use.

```
<!–integer literal with value 3-->
<lit>3</lit>
<lit value="3" />

<!–constant with name and value-->
<const name="pi" datatype="real">3.14</const>
<const name="pi" datatype="real" value="3.14" />
```

C. Variables and Domain

Variables form the primary components of any constraint problem definition. They are named entities in a constraint model which take one or values Solving constraint satisfaction as well as optimization problems revolve around assigning optimum and permissible assignments to one or more variables involved in the model.

Variable definition consists of:

- Name: It acts as its identifier, it is used as a reference elsewhere in the model definition. Name should be unique within a single model definition.
- Domain: It defines the range or set of permissible values which the variable can be assigned to. Domain can be represented explicitly by listing all the allowed values or implicitly by providing range through indexers of functions.
- Datatype: The datatype of the underlying variable which can be integer, boolean, real or string

Name is mandatory for a variable whereas domain and datatype are optional. If a datatype is not mentioned in the specification then it can assumed to be string.

Variable specification with name, datatype and domain

```
<variables>
<!–boolean variable-->
    <variable name="A" datatype="bool" />
<!–integer variable with allowed value:3, 5, 7
and 14-->
    <variable name="B" datatype="int">
      <domain>
        <set type="int">
          <lit>3</lit>
          <lit>5</lit>
          <lit>7</lit>
          <lit>14</lit>
        </set>
      </domain>
    </variable>
<!–integer variable with allowed value:2 to 10
-->
    <variable name="C" datatype="int">
        <domain>
          <set mode="sequence">
            <lower>2</lower>
            <upper>10</upper>
            <delta>2</delta>
          </set>
        </domain>
      </variable>
</variables>
```

It should be noted that Variables can be expressed as a function of other variables too.

D. Subset Definition

Subsets are basically lists of literals, constants or any expression which can be evaluated to generate a list of elements of the same type. They are widely used in constraint specification when a group of data is to be used. They enable a user to concisely represent a group of data. Subsets can be either explicitly or implicitly denoted. In XML or JSON explicit subsets can be represented as a set of literals or constants enclosed in a pair of opening and closing set tag.

[1, 2, 3, 4, 5]	[1.52, 2.9, 3.5, 4.22]
<set name="list" datatype="int"> <lit>1</lit> <lit>2</lit> <lit>3</lit> <lit>4</lit> </set>	<set name="set1" datatype="real"> <lit>1.52</lit> <lit>2.9</lit> <lit>3.5</lit> <lit>4.22</lit> </set>

A generated or implicit list does not list all the contents but includes specification for the interpreter/solver to expand the list. They are generally computed using integer of real values and are represented with tags: <sequence> or <series>. They consist of lower bound, upper bound, delta value and an optional function tag which is used to iteratively be applied to the lower bound. If delta value is not mentioned it should be assumed to be 1. Example representation of sequence and series using XML notation is as follows:

[2, 4, 6, 8, 10]	[2, 4, 8, 16]
<seq datatype="int"> <lower>0</lower> <upper>10</upper> <delta>2</delta> </seq>	<series> <lower>2</lower> <upper>20</upper> <delta>2</delta> <function type="product"> <var>lower</var> <var>delta</var> </function> </series>

E. Tables

A table is used to contain a list of tuples each of which represents a permissible assignment over a set of variables. It is something very similar to a lookup table. The solver can use this table to lookup list of permissible values for a single variable or set of variables. A table consists of an assignment part which can be represented using assignments.

For example if a model problem has a variable X of type Real, and it can be assigned any values from 4.5, 8.9 or 12. Then it can be represented using XML notation as follows:

X = 4.5, 8.9 or 12	(X,Y) = { (3,3), (1,5) (4,5) }
<table name="tableX"> <variables> <var>X</var> </variables> <tuples> <lit>4.5</lit> <lit>8.9</lit> <lit>12</lit> </tuples> </table>	<table name="tableXY"> <variables> <var>X</var> <var>Y</var> </variables> <tuples> <tuple> <entry key="X" val="3"/> <entry key="Y" val="3"/> </tuple> <tuple> <entry key="X">1</entry> <entry key="Y">5</entry> </tuple> <tuple> <entry key="X">4</entry> <entry key="Y">5</entry> </tuple> </tuples> </table>

We can also represent paired constraint on multiple variables using the table notation, that is X, Y can have values 3,3 or 1,5 or 4,5 as shown in the table above.

F. Iterators and Index Variables

An iterator as the name suggests iteratively applies a computable or Boolean expression on one or more index variables which are part of iterators. An iterator primarily consist of three components:

- Index Variable: An iterator runs on domain values typically bounded by a lower and an upper value, an iterator can have one or more number of variables
- Iterator Function: They can be either arithmetic like: sum, product etc. or they can be logical application of a Boolean expression like: forall, exists, existslessthan etc.
- Template Expression: This can be represented in the form of an expression tag which can further consist of one or more functions acting on variables or just plain variables as follows:

```
<expression>
    <function type="product">
      <lit>2</lit>
      <var>i</var>
    </function>
</expression>
```

Iterator for summation: $\sum_{n=1}^{10} 3 * n$

```
<iterator type="sum">
  <indexers>
    <index name="n" datatype="int">
      <lower>1</lower>
      <upper>10</upper>
      <delta>1</delta>
    </index>
  </indexers>
  <expression>
    <function type="product">
      <lit>3</lit>
      <var>n</var>
    </function>
  </expression>
</iterator>
```

Iterator can also be used for representing universal quantification(forall) or existential quantification(there exists): Example: For all natural numbers n = 1 to 10, 2·n == n + n

```
<iterator type="forall">
  <indexers>
    <index name="n" datatype="int">
      <lower>1</lower>
      <upper>10</upper>
      <delta>1</delta>
    </index>
  </indexers>
  <expression>
    <function type="equal">
      <function type="product">
        <literal>2</literal>
        <variable>n</variable>
      </function>
      <function type="sum">
        <variable>n</variable>
        <variable>n</variable>
      </function>
    </function>
  </expression>
</iterator>
```

Following are the type of iterators which can be defined using the 'type' property:

- Sum: This function computes iterative summation of variables/constants within the expression tag using the iterator.
- Product: It computes iterative product of variables/constants within the expression tag using the iterator.
- Forall: Evaluates the boolean template expression for each of the index-variable domain substitutions, in short it returns 'true' iff the template expression evaluates to true for all substitutions.
- Exists: Similar to forall, but it will evaluate to true if and only if at least one domain-value substitution yields a true evaluation for the template expression.
- ExistLessThan: Similar to exists except for the change that function checks for existence of at least one domain-value substitution that will yield a value less than
- ExistsGreaterThan: Similar to ExistLessThan but checks for at least one value greater than
- ExistsEqual: On the same lines as Exists with addition of the fact that it evaluates to true iff at least one index-variable domain substitution equals to desired value.
- SumIf: Evaluates to true if the iterative sum of index-variable domain equates to a defined value
- ProductIf: Evaluates to true if the iterative product of index-variable domain equates to the desired value.

III. CONSTRAINTS SPECIFICATION

Constraints form the heart of CP modeling, they are basically set of rules which govern limitations on permissible values that can be assigned to a variable or a group of variables. On usual occasions there can be multiple ways to express a constraints. They can be broadly segregated based on their types of operators and also the context in which they are used as logical functions, relational expressions

A. Logical Functions and Expressions

A number of constraint specification can be expressed through the use of logical functions and expressions. They can be depicted as operators which operate on one or more operands and based on the type of operator produce a specific output. They can be either unary (operating on a single operand/entity) or binary (operating on two operands or entities) in nature. Logical functions are also referred to as Boolean functions.

Logical expressions can be represented as an n-ary tree which consists of functions and its operands in our XML notation. The solver can then perform an infix, postfix or prefix traversal of the binary tree and expand the underlying

expression to use in computation or problem solving. For example if we have a logical conditional expression which states that if a certain variable A is true then either B is true or E is true, this can be expressed as follows using XML:

A -> (B I E)

```
<constraints>
    <constraint name="c1">
      <function type="conditional">
        <var>A</var>
        <function type="or">
          <var>B</var>
          <var>E</var>
        </function>
      </function>
    </constraint>
  </constraints>
```

As you can see the logical functions take the form of a function which can contain one or more variables or one or more other functions. The interpreter of solver using the above XML constraint specification can infer the type of function based on the type property. The type of logical functions can be NOT, AND, OR, XOR, Conditional and Equivalence. It should be noted that while representing a unary logical/Boolean expression using the XML notation the mode of the function tag should have a mode property which should be set to unary. If mode property is not set then it is assumed to be binary in nature. A logical function with unary operator like: either A is false or B is false if D is false can be depicted as follows:

!d -> (!a I !b)

```
<function type="conditional">
      <function mode="unary" type="not">
        <var name="d"/>
      </function>
      <function type="or">
        <function mode="unary" type="not">
          <var name="a"/>
        </function>
        <function mode="unary" type="not">
          <var name="b"/>
        </function>
      </function>
</function>
```

B. Relationsl Functions

Relational functions can be represented in a format very similar to logical expression that is using n-ary tree to depict the relations of operators and operands through XML. They are of types: greater-than, less-than, equal, not-equal, greater-than-equal and less-than-equal. For example consider a relations function expression: variable t3 is greater than or equal to variable t2 and t3 is greater than variable t5. This can be represented in XML as:

t3 >= t2 & t3 > t5

```
<function type="and">
    <function type="greaterthanequal">
      <var>t3</var>
      <var>t2</var>
    </function>
    <function type="greater">
      <var>t3</var>
      <var>t5</var>
    </function>
  </function>
```

It should be noted that, ordering of operands of a relational function should be consistent in the XML notation. That is for expression: t3 > t5, the variable t3 should sequentially come before specification of variable t5 as shown in the above snippet.

IV. GLOBAL CONSTRAINTS

Representation of constraints covering multiple variables can be done in a simple and intuitive way through the use of global constraints. In simple terms it enables the user to express multiple constraints spanning across multiple variables at a single place which would otherwise take lengthy expressions, multiple variable declaration etc. There are number of global-constraints functions which are defined as follows:

A. Alldifferent Constraint

It is a global constraint which evaluates to a boolean expression which evaluates to true if and only if all variables covered under the constraint have a different value within their domain. Alldifferent is very much like a function wherein all the participating parties need to have different values in order for the constraint to be satisfied. That is if there are N number of variables belonging to a domain D which has d number of possible values, than each and every variable should have a one to one mapping with domain values and no two variable can take the same value. Alldifferent constraint can be easily depicted with our XML specification by simply wrapping the underlying variables or indexers using an alldifferent tag as follows:

Subset row: 1 to 10
Subset column: 1 to 10
Map (column, row): cells
Alldifferent(cells)

```
<alldifferent>
```

```
  <variables>
    <var name="cells"/>
  </variables>
</alldifferent>
```

As stated in the above definition there are couple of subset named row and column which can have values from 1 to 10. Cells is a variable of type Map which is combination of row and column type, that is Map can consist of paired values of row and column. For example map can have three tuples with values: [(1, 2) (2.3) (3, 3)]. Alldifferent states that each tuple within Map should have unique paring that is no two tuples can have values like (1, 2) and (1, 2).

B. Allsame Constraint

Allsame constraint is very similar to alldifferent in the sense that it is a Global constraint with the exception that the model-variable inputs must all assume the same value. This function is apt for situations wherein you want a set or group of variables to be equivalent to the same value. The XML listing for allsame constraint is same like alldifferent except for the use of allsame XML Tag.

C. Domain Cardinality Constraint (DCC)

DCC is a type of constraint with which one can place an upper of a lower bound on a set of variables that can simultaneously be assigned a value from a particular domain. For example if we have integer variables A, B, C and D each having domain: [1, 2, 3, 4, 5]. Then through the use of DCC we can restrict certain values to be assigned to certain number of variables, for example value 2 can only be assigned to maximum of three variables or value 5 can be assigned to only one variable. DCC can have variations in terms of greater than equal to, less than equal to, greater than, less than and equal to, which states the permissible numbers of assignments to a variable from domain. Using XML Notation we can represent DCC constraints as follows:

Subset Task: 1 to 10
Subset StartTime: 1, 2, 3, 4
Map (Task, StartTime): TaskTime

DomainCardinalityConstraint-Less Than Equal (TaskTime, 4)

```
<dcc type="greaterthanequal">
  <variable>
    <var name="TaskTime"/>
  </variable>
  <upper value="4" />
</dcc>
```

As stated above Task is a set of numbers tasks from 1 to 10 and start time can be 1, 2, 3 or 4. TaskTime is mapping of a particular Task to a particular start time. The above XML dcc states that at most four tasks can be started at any given time. DCC constraints of lessthanequal, lessthan and greater than can be depicted in a similar way by assigning them of the proper type.

V. OBJECTIVE

Objective of a constraint processing model is description of a function that should be maximized or minimized bounded by the domains and constraints of the variables which form part of the objective function. It is a way through which constraint model issues directives to the solver about what it seeks to solve. Objective can be broadly classified into two categories: constraint satisfaction problem, that is to find permissible value for variable or set of variables within the domain and constraints of the model and category two is constraint optimization problems in which consumer wants to find maximum or minimum value to variables in congruence to the constraints.

Satisfaction problems can be illustrated using an objective tag and a satisfy tag within which all variables for which values is to be computed can be listed as shown below:

Objective:
Satisfy: X, Y

```
<objective>
  <satisfy>
    <var name="x"/>
    <var name="y"/>
  </satisfy>
</objective>
```

As for constraint optimization problem, a consumer can seek optimization either of variables or relational or arithmetic expressions involving multiple variables. This can be modeled using XML tag as follows:

Objective:
Maximize: a
Minimize: x + y

```
<objective>
  <maximize>
    <var name="a"/>
  </maximize>
</objective>

<objective>
```

```
  <minimize>
    <function type="sum">
      <var name="x"/>
      <var name="y"/>
    </function>
  </minimize>
</objective>
```

VI. Model Solution

Model solution is defined as assignment of variables that satisfies all the constraint listed in the model definition. Finding a solution to a constraint satisfaction problem is reducing the sample space to locate a single assignment to the variables of the model.

Puzzle:

A= true
B = false
C = true

```
<solution>
  <model name="Puzzle">
    <variables>
      <var name="A" value="true"/>
      <var name="B" value="false"/>
      <var name="C" value="true"/>
    </variables>
  </model>
</solution>
```

A solution to a constraint model can be depicted by using a <solution> XML tag which will further contain assignments for variables contained in the model tag and the solution values for variables can be set to the value property of the var tag. It should be noted that it is not unusual for a solver not to find a solution at all in which case an empty tag of <solution></solution> can be returned to the consumer.

There are scenarios for certain solvers wherein they need to convey more information in addition to the value of model variables. For example a solver was unable to find the solution to a problem even after scanning through the entire search space, whereas it might also be that it returns the first solution and skips scanning the whole search space. This information can be conveyed to the consumer by setting 'searchspace' property of solution XML tag to any of these following values:

- Complete: If at least one solution was found and the solver terminated after exploring the entire search space of the constraint model.
- Incomplete: If at least one solution was found but the solver terminated after exploring only partial search space.
- Unsatiable: If no solution was established although the solver scanned the entire search space of the model.
- Unknown: If no solution was found and the solver had to prematurely exit the process after partial exploration of search space.

VII. LINEAR FUNCTIONS

Linear functions occur quite commonly in constraint processing problems. They are a very simple and elegant way to represent constraints, linear functions are first degree polynomial of the form:

$$F(x) = mx + c$$

For example, consider the following linear programming problem in standard form:

$$Maximize = x + y \ (x, y > 0)$$

Subject to constraints:

$$5x - y <= 9$$

$$2x + y <= 12$$

This can be represented as follows using our notation of XML:

```
<model name="linearprogramming">
    <var>
      <var name="x" datatype="int">
        <domain>      <!--variable domain-->
          <function mode="unary"
type="greaterthan">
            <constant>0</constant>
          </function>
        </domain>
      </var>
      <va name="y" datatype="int">
        <domain>      <!--variable domain-->
          <function mode="unary"
type="greaterthan">
            <constant>0</constant>
          </function>
        </domain>
      </var>
    </var>
    <objective>       <!--definition of
objective-->
      <maximize>
        <function type="sum">
          <var>x</var>
```

```
          <var>y</var>
        </function>
      </maximize>
    </objective>
 <constraints>
  <!--constraint definition-->
  <constraint name="c1">
    <linearequality>
      <bound>9</bound>
      <term>
        <var name="x" />
        <coefficient>5</coefficient>
      </term>
      <term>
        <var name="y" />
        <coefficient>-1</coefficient>
      </term>
    </linearequality>
  </constraint>
  <constraint name="c2">
    <linearequality>
      <bound>12</bound>
      <term>
        <var name="x" />
        <coefficient>2</coefficient>
      </term>
      <term>
        <var name="y" />
        <coefficient>1</coefficient>
      </term>
    </linearequality>
  </constraint>
</constraints>
</model>
```

VIII.MODELING KEYWORDS

Mathematical Functions	
sum	sqroot
difference	negation
multiply	inverse
division	cuberoot
modulo	powerof
ceil	cos
floor	tan
log	cosec
log10	sec
sin	cot
exp	abs

Logical Functions	
Equal (= =): Evaluates to true if all operands are equal in value.	equality (= = =) Evaluates to true if all operands are equal in value and in datatype.
Not (!) Performs Boolean negation of the operator	Notequal (!=) Evaluates to true if pair of operands are not equal
And (&) Evaluates to true if both operands are true else it is false	Or (|) Evaluates to false if both operands are false else it is true
Exclusiveor (^) Evaluates to true only if both operands have different Boolean value, is false otherwise.	Conditional (->) It represents conditional if-then function
Equivalence (<->) It represents a double implication. For example if P <->Q it is equal to true if P and Q are both true or if P and Q are both false; otherwise the double implication is false	

Relational Functions	
lessthan	greaterthan
lessthanequal	greaterthanequal

XML Elements	
Set	XML tag represents element of type set.
Seq	XML tag represents element of type sequence.
Model	XML tag represents root element which contains variables, constraints and other entities.
Var	Element tag describes a variable.
Function	Element tab is used to depict a function if type unary, binary or n-ary.
Lit	Represents a literal tag to mention a literal value.

Lower	This tag is used to mention lower bound value for an iterator or sequence.
Upper	This tag is used to mention upper bound value for an iterator or sequence.
Delta	Tag to set delta seed value used by iterators or sequence to increment/decrement
Iterator	XML tag to create iterator entity.
Expression	XML tag to depict an expression which is usually child of a parent entity.
Indexers	Tag to represent group of indexers on an iterator.
Index	Index element to declare integer indexers which is used to iterate in sequence or an iterator.
Tuple	Tag to represent tuple which represents pairing of values contained in a table entity.
Table	XML tag to depict a table entity which includes tuples and entries.
Objective	Tag to specify the objective model, variable or function.
Satisfy	Specify that the underlying objective is to find satisfying assignment to variables
Maximize	Tag to direct the solver to maximize the enclosed variable or expression.
Minimize	Tag to direct solver to minimize the enclosed variable or expression.
Alldifferent	Global constraint for all different Boolean constraint
Allsame	Global constraint for allsame Boolean constraint
Dcc	Tag to enclose a domain cardinality constraint (global constraint)
Primitive Data Types	
boolean	real
string	integer

IX. RELATED WORK

There is a growing community of people and organizations working towards development and betterment of a solver independent model specification language.

The St Andrews Constraints Group, along with their colleagues at Cork, Dundee and York, offer a complete constraint modelling toolchain that, starting with an abstract specification of a constraint problem, performs the modelling and solving phases efficiently and automatically. But their toolchain is tightly coupled to their specification which is in Essence format. Although Essence is capable to depict wide variety of constraints and variables, it still lacks in the ease of use and wide acceptance of the underlying format.

X. FUTURE WORK

A common language for modeling constraint opens up a horizon to a whole new playing field for constraint programming. One can deploy a network of constraint solvers on varied technologies and having different search tactics which can be executed in parallel which will enable solvers to compete as well as collaborate to find solution to the problem. Search agents can communicate itermittent solutions, newly discovered sample spaces as well as false positive domain for the solution and this information can be shared amongst them to find efficient, accurate and fast solutions to constraint problems.

Future work can be done on the development of a visual tool which enables an end user to define model and constraints through a graphical user interface equiped with drag and drop functionality, intuitive controls which would auto-generate the XML/JSON specifiation. The implementation of a master agent service deployeable on an Internet is also an interesting task, it will accept model specification is XML or JSON format and route it to various agents which will be equipped with solvers, each solvers will seek a solution based on their strategies and techniques. Solvers can either all work on the same specification and compete on a race to the finish line or they can each be assigned part of the model and find local/partial solution to the problem which can then be combined to solve the actual problem.

XI. CONCLUSION

CP is a time-tested and proven methodology for finding solution to complex optimization or decistion problems. Their utility is ever occurring and evident in the domain of transportation, assembly line scheduling, manufacturing, construction, landscaping etc. But a universal format of language has not been established or widely supported. The proposed XML-JSON modeling approach is initial work for setting up a baseline for a universally accepted format of constraint modeling which is independent of a solver, sort of a common language which solvers can understand to exchange problem definition and solution. Constraint modeling using XML enables robust, expandable and human readable format

for constraints. The format is also API/Technology friendly as XML and JSON are widely . It will lead to development of problem-modeling consumers to communicate with solvers which are capable of solving specific families of problems. In turn, these agents possess access to high-end computing infrastructure are capable of solving the problem faster than what the client/consumer will take.

ACKNOWLEDGMENT

I am grateful to Dr. Todd Ebert for providing his guidance throughout the course of my research. His feedback and suggestions were very valuable in the progress and completion of this paper and the corresponding project.

REFERENCES

[1] Ralph Becket, Specification of FlatZinc, Version 1.6 http://www.minizinc.org/specifications.html

[2] Rapha¨el Chenouard, Laurent Granvilliers and Ricardo Soto "Model-Driven Constraint Programming" PPDP'08, July 15–17, 2008, Valencia, Spain.

[3] Rina Dechter, Constraint Processing. Morgan Kaufmann, 2003

[4] Constraint Modelling (2015, May 7) [Online]. Available: http://constraintmodelling.org/

[5] Dr. Todd Ebert, Constraint Programming using Clara Chapters 1, 2 and 3.