Instance Transformation for Declarative Solvers
Or: Instant Model Finders

Megan O'Connor and David Mitchell
Computational Logic Laboratory
School of Computing Science
Simon Fraser University
Burnaby, B.C. V5A 1S6 CANADA
{mitchell,megano}@cs.sfu.ca

***Abstract*—We describe a method and prototype tool for purely declarative creation of "solvers" for a wide range of problems where instances are presented as strings. Examples include model finders for logics of moderate expressive power. The method exploits existing specification-based declarative problem-solving systems ("model-and-solve" systems), adding a front-end tool to declaratively map problem instances in arbitrary form into system-specific instance formats. We illustrate application of our prototype tool with examples including graph problems and non-classical propositional logics.**

I. Introduction

Systems for solving combinatorial problems based on declarative specifications are growing in variety, number, and range of effective application. Examples include systems using the following specification languages: the Alloy software modelling language [1], algebraic modelling languages used in mathematical programming such as Zimpl [2], constraint modelling languages used in combinatorial optimization such as MiniZinc [3], and knowledge representation languages such as used in the IDP System [4], Enfragmo [5], and Answer Set Programming systems such as clasp [6] and DLV [7].

The specification languages, instance (or data) file formats, and associated terminology for these tools can differ considerably, depending on the intended application problem, target user community, or developer community. All, however, provide users with a similar capability, which is to solve combinatorial problems by writing high-level declarative problem specifications rather than executable code.

Use of this sort of system is conceptually simple, as illustrated in Figure 1. The user writes a problem specification in the declarative language of the system, the solver takes as input the problem specification and a problem instance, and outputs a solution for the instance (if it can find one). For example, for the problem of Graph Colouring, the problem specification says (in syntax of the relevant language) that every node of the graph must be assigned one of the available colours, and that no pair of adjacent vertices are assigned the same colour. Such a specification can easily be written in any of the languages mentioned above. Then, the specification and a particular graph are given as input to the system, which will try to construct a proper colouring.

The declarative nature of these systems suggests they can be used by workers who are neither programmers nor experts in combinatorial problem solving or optimization. Because they are used without coding, they are sometimes call "model-and-solve" systems. It is reasonable to expect that, in the near future, cloud-based model-and-solve systems will provide practical combinatorial problem solving not only to those without programming or combinatorial expertise, but also without local computational infrastructure, and at modest cost.

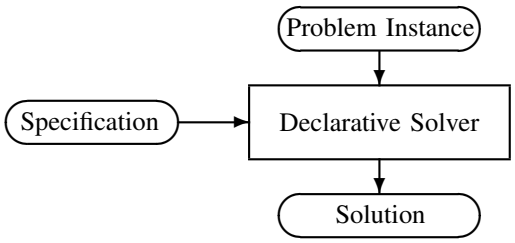


Fig. 1. Declarative Problem Solving Systems

However, the above description of using such a system left out an uninteresting but essential step: the problem instance must be put into the input format for the solver to be used. With few exceptions, in serious applications this requires writing a program. Mundane though this coding typically is, it prevents the system from being purely declarative, and from being used by non-programmers.

The purpose of this paper is to introduce a method, and a prototype tool, that addresses this problem. The tool, used as a front-end for an existing declarative solving system, allows for purely declarative problem solving, by providing a means of declaratively mapping instance data from text files, in whatever form they are obtained, to the instance format of the solver.

As we will see, our solution to the mundane problem will allow us to do something much more interesting as well. Suppose that someone has presented us with instances of a problem described in some unusual logic or specification language. Or, suppose that we have designed a non-standard logic which we think will be useful for some application, and we want to experiment with it. Building a model finder can take considerable programming effort. With the method described here, one can turn a general purpose solver, such as an Integer Programming solver or other constraint solver, into

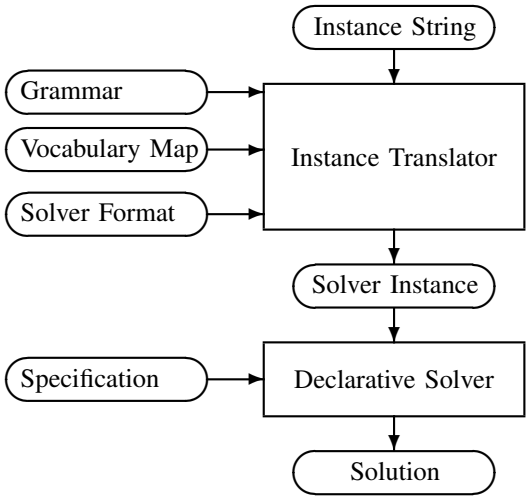


Fig. 2. Scheme for using the instance-translation front-end with a general-purpose declarative specification-based solver.

a model finder for a wide variety of logics with only a page or so of declarations, and no code. Ergo this paper's sub-title.

In the following section, we give some formal preliminaries. We describe the overall method in Section III, with further details in Section IV, and give some examples in Section V. We describe our prototype tool in Section VI and the method of generating solver-specific inputs in Section VII, with a brief concluding discussion in Section VIII.

II. Preliminaries

We assume the reader is familiar with basic propositional and first order logic. Our propositional formulas have atoms P_i, connectives $\wedge$, $\vee$, $\neg$; and parentheses $(,)$. First order formulas have predicate symbols P_i, functions symbols f_i, variable symbols x_i and quantifier symbols $\forall$ and $\exists$, as well. Formulas are defined by the usual inductive construction, with the standard semantics. The semantics of propositional formulas are defined in terms of truth assignments to the propositional atoms. The formal semantics of first order formulas are defined in terms of structures (or interpretations). A vocabulary is a set of function and relation symbols, each with an associated arity. (Constant symbols are zero-ary function symbols.) A structure $\mathcal{A}$ for vocabulary τ (a τ-structure) is a tuple consisting of a set A, called the universe of $\mathcal{A}$, a k-ary relation over A for each k-ary relation symbol of τ, and a k-ary function on A for each k-ary function symbol of τ. If $\mathcal{A}$ is a τ-structure, and P a predicate symbol of τ, then we write $P^{\mathcal{A}}$ for the relation in $\mathcal{A}$ corresponding to P, called the interpretation of P by $\mathcal{A}$. A good references for details is [8].

III. Method Description

The function of the front-end tool is to map a string, which describes an instance of a problem, into the instance (or data file) format for a chosen declarative problem solving tool. A user should be able to describe this mapping, purely declaratively, for a wide variety of languages, including languages

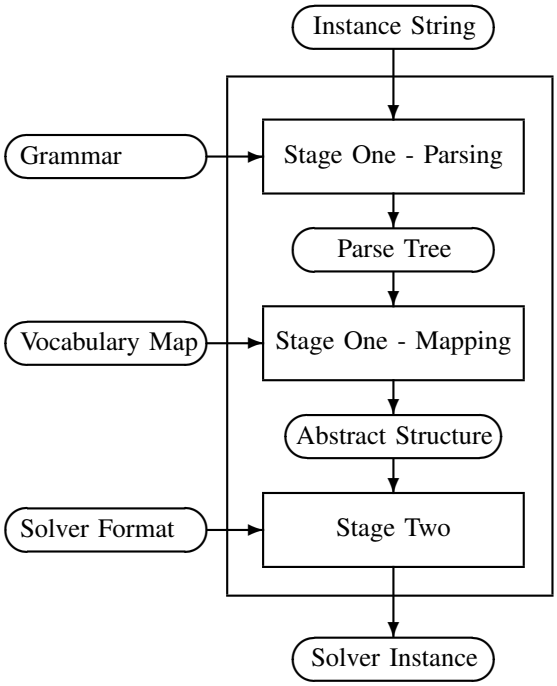


Fig. 3. Internal scheme of the instance translator.

and systems the tool developer may not have anticipated. To support any reasonable level of generality, we require an abstraction of what a "problem instance" is. Application of the tool is illustrated in Figure 2. The user must provide the following four items:

1) A *grammar* for the language in which problem instances are described, which the tool uses to parse the input string.
2) A *vocabulary map*, which describes a mapping of semantically significant syntactic features to the semantic vocabulary of the abstract instance representation.
3) A *solver format* description, which defines a mapping from the abstract instance representation to a solver-specific instance format.
4) A *problem specification*, in the specification language of the chosen solver.

While superficially very different, all of the specification-based solvers mentioned above, and many others, can be seen to solve the same general abstract problem, and thus formalized in a uniform way. This formalization makes it possible to separate the problem of recognizing a string as a description of a problem instance from the presentation of the instance in a solver-specific format: We translate first from the string to a generic abstract representation, and then to the desired solver format. The abstract structure, here, plays the role of an interlingua in a multi-language translator.

The input instance, for solvers of the sort we are considering, is always a collection of finite functions and relations. For example, a graph consists of a unary relation (the set of vertices) and a binary relation, (the set of edges). A weighted graph has, in addition, a weight function mapping edges to

numbers. The solution to be output is a related collection of functions and relations. For example, a sub-set of the edges of a given graph (e.g., a minimum-weight spanning tree or matching), or a function from the vertices to colours. The problem specification describes, in some formal language, the relationship between the the input relations and the output relations which constitute solutions. This specification is written using a vocabulary of symbols which include symbols denoting the input and output relations. We may formalize this quite generally as follows.

Because an instance of a problem is a collection of finite functions and relations, it can be treated formally as a finite structure, where "structure" here is used as in mathematical logic. Similarly, the solution is a structure. To write a specification for the problem, we use a vocabulary consisting of function and relation symbols denoting elements of the instance, and elements of the solution, possibly together with other symbols having standard meanings such as those used in arithmetic expressions.

Suppose that vocabulary σ is a proper subset of vocabulary τ, and $\mathcal{A}$ be a σ-structure. If $\mathcal{B}$ is a τ-structure which agrees with $\mathcal{A}$ on σ (that is, for every $P \in \sigma$, $P^{\mathcal{A}} = P^{\mathcal{B}}$), then we say that $\mathcal{B}$ is an expansion of $\mathcal{A}$ to τ, and $\mathcal{A}$ is the reduct of $\mathcal{B}$ on σ. Now, if σ is our chosen instance vocabulary, then instances are σ structures. If γ is our chosen solution vocabulary, let $\tau = \sigma \cup \gamma$. If σ-structure $\mathcal{A}$ is a problem instance, then any structure $\mathcal{B}$ that is an expansion of $\mathcal{A}$ to the combined instance and solution vocabulary τ, is a structure consisting of an instance together with a "possible solution".

Now, such a specification can be viewed formally as a sentence, with vocabulary τ, of some quantified logic that defines solutions to the problem by defining the class of τ-structures which consist of a problem instance together with one of its solutions. A problem instance $\mathcal{A}$ has a solution if and only if there is an expansion of $\mathcal{A}$ to τ that satisfies specification formula S. The task of the solver is to find such an expansion if there is one. (Suppose the specification is a formula ϕ. There is an expansion of $\mathcal{A}$ that satisfies ϕ if and only if $\mathcal{A}$ is a model of the (second-order) formula $\exists \vec{R} S$, where $\vec{R}$ is the vocabulary symbols in τ. Hence [9] used the term "Model Expansion" for the underlying formal task of specification-based solvers.)

Example 1: Consider the problem of finding a proper colouring of a graph G with colours from a set C. An instance consists of a set of vertices, a set of edges, and a set of colours. Let our "instance vocabulary" be $\sigma = \langle V, E, C \rangle$, where V and C are unary relation symbols (they denote sets), and E is a binary relation symbol. A solution is a function mapping colours to vertices, and satisfying certain properties. Let our "solution vocabulary" be $\gamma = \langle Col \rangle$, where Col is a binary function symbol (which will map vertices to colours). A specification for the Graph Colouring problem is a formula ϕ with the property that a τ-structure satisfies ϕ if and only if it is a properly coloured graph. To illustrate:

$$\overbrace{\underbrace{(V, C; E^{\mathcal{A}}}_{\mathcal{B}}, Col^{\mathcal{B}})}^{\mathcal{A}} \models \phi$$

(Here, $\mathcal{A}$ is the graph and set of colours, and $\mathcal{B}$ is the expansion with a node colouring function.)

The following formula ϕ has the required property, and thus constitutes a specification for Graph Colouring.

$$\begin{aligned} &\forall x \exists c[Col(x,c)] \\ \wedge &\forall x \forall c[Col(x,c) \supset \neg \exists k((c \neq k) \wedge Col(x,k))] \\ \wedge &\forall x \forall y[E(x,y) \supset \forall c(\neg(Col(x,c) \wedge Col(y,c)))] \end{aligned}$$

Several of the specification languages we mentioned in the introduction are explicitly described as logics, albeit with non-standard syntax and/or extensions beyond textbook-style first order logic for practical convenience in modelling real problems. Others, while normally not thought of or described as formulas of a logic, are in fact easy to view as syntactic variants of first order logic, again with an number of extensions, such as type systems and built-in arithmetic. (Illustrations of this can be found in [10] and [11].) So, this formalization is indeed applicable to real systems.

Given this formalization, we may accomplish our translation of instance description strings to solver input formats in two stages, as illustrated in Figure 3. Stage one maps the string to a structure. Stage two maps the structure to the file format of the desired solver. Stage one requires a grammar for parsing the strings, and a "vocabulary map", which defines the vocabulary of the instance structure, and a map from semantically meaningful syntactic elements from the parse tree to the instance structure.

In the following sections, we will describe this process in further detail, using propositional logic to illustrate. That is, we suppose we have a general-purpose, declarative-specification based solver, and want to use it to find satisfying assignments to formulas of propositional logic. As a running example, we use the following small formula.

Example 2: Let F be the formula

$$((p \vee q) \wedge (\neg p))$$

.

IV. Mapping Strings to Structures

We want to map strings that describe problem instances to structures. The relations in these structures should make the semantic content of the strings explicit. Any reasonable language will have syntactic constructs corresponding to the semantically meaningful properties of the problem instance. To identify these syntactic constructs, we need to parse the formula, and the relevant syntactic constructs will be reflected in the parse tree produced by the parser. Then, we need to map the relevant features in the parse tree to the desired structure.

A. Step One: Parsing

In the first step, a parse tree is constructed using a generic parser, which takes as input the grammar and a string, and parses the string in accordance with the grammar. For our example problem of propositional logic, here is a suitable grammar.

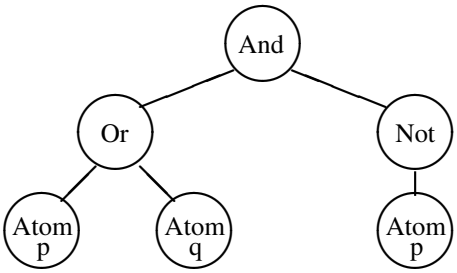
Example 3:

$$\begin{aligned} Formula &\rightarrow Atom | \text{“(”} And \text{“)”} | \text{“(”} Or \text{“)”} | \text{“(”} Not \text{“)”} \\ And &\rightarrow Formula \text{“}\wedge\text{”} Formula \\ Or &\rightarrow Formula \text{“}\vee\text{”} Formula \\ Not &\rightarrow \text{“}\neg\text{”} Formula \\ Atom &\rightarrow \_character[lower] \end{aligned}$$

The rule $Formula \rightarrow Atom | \text{“(”} And \text{“)”} | \text{“(”} Or \text{“)”} | \text{“(”} Not \text{“)”}$ describes which strings are considered Formulas, and indicates that all Formulas (except atoms) are enclosed within parentheses. The rule for Atom indicates that an atom is a sequence of lower case characters.

Parsing the instance formula according to the grammar generates a parse tree for the instance. Each node corresponds to an application of a grammar rule, and is labelled with that rule.

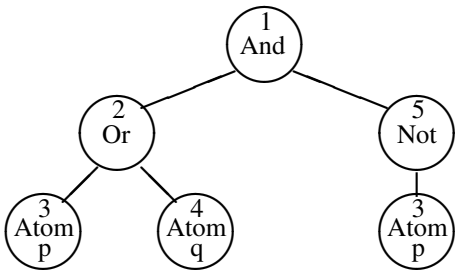
Example 4: The parse tree P of F is



B. Step Two: Structure Construction

For our example problem of propositional logic, we want to map propositional formulas to structures. The relations in these structures should make the semantic content of the formula explicit. This semantic content is (in accordance with the usual recursive definition of satisfaction of a propositional formula), the relationship between distinct sub-formulas. We will assign an identifier for each sub-formula, and the elements of the relations in our structure will be these identifiers. Nodes are assigned identifiers starting with 1 for the root of the parse tree and assigned in depth-first-search order incrementally through the tree, except that leaves, which correspond to terminals in the grammar are treated differently. If two terminal nodes correspond to the same semantic element (in this example, they are atoms) then the nodes are assigned the same identifier.

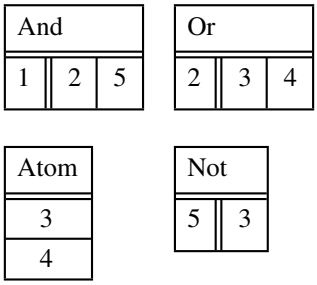
Example 5: The nodes of parse tree P with unique identifiers assigned.



Mapping the parse tree to a structure amounts, essentially, to placing semantically similar sub-trees in relations. The structure for a formula will have a relation for each of the three connectives $\wedge$, $\vee$ and $\neg$, and also a set (unary relation) identifying the atoms of the formula. Each tuple in the relation $\wedge$ represents a sub-formula which is a conjunction.

We can visualize a structure as a collection of tables, one for each relation in the structure (and each semantically significant syntactic structure).

Example 6: In the structure corresponding to $F = ((p \vee q) \wedge (\neg p))$, the relation for $\wedge$ will have a triple representing the fact that F consists of conjunction of two sub formulas: $\langle F, F_1, F_2 \rangle$, where F_1 and F_2 are identifiers for the sub-formulas $(p \vee q)$ and $(\neg p)$. The structure for formula F is:



The parse tree is mapped to the abstract structure by mapping each node of the tree to a row of a table based on the vocabulary map. Which table a node is mapped to is based on the type of semantic element the node represents and is determined by the grammar rule recorded by the node. The columns of a table contain the identifiers of a node and its children.

V. Examples

Our method can be used for a variety of problems ranging from graph problems to non-classical propositional logics and beyond. Here we give some simple examples to demonstrate some features of our method. For each problem, we will give a small input string as an example, and a grammar for parsing the strings. We also give a problem specification, written in first-order logic, which defines solutions to the problem instances using the vocabulary corresponding to an actual vocabulary map used for our prototype. (We give the specifications in a modest extension of first-order logic so the reader does not have to be familiar with the specification language of any particular system. Suitable specifications are easy to write in all the languages mentioned in the introduction to the paper.)

A. Graph Problems

Here is an example of a graph G, as represented in the DIMACS graph format for clique and colouring problems [12]. The first line gives the numbers of edges and nodes. Each line beginning with an 'n' gives a node number and it's weight. Each line beginning with an 'e' gives an edge.

```
p edge 4 3
n 1 5
n 2 7
n 3 13
n 4 1
e 3 2
e 2 1
e 1 3
```

The DIMACS graph format is described by the following grammar:

$$\begin{aligned} Problem &\rightarrow \text{“}p\text{”}\ \text{“}edge\text{”}\ Num\ Num\ Graph \\ Graph &\rightarrow Edg|Node|Edg\ Graph|Node\ Graph \\ Edg &\rightarrow \text{“}e\text{”}\ Vtx\ Vtx \\ Node &\rightarrow \text{“}n\text{”}\ Vtx\ Num \\ Vtx &\rightarrow \_integer[1:50] \\ Num &\rightarrow \_integer[0:100] \end{aligned}$$

With a corresponding vocabulary mapping it can be used to find a clique of large weight in graph G using the specification:

$$\begin{aligned} \forall x \forall y[&(Clique(x) \wedge Clique(y)) \supset \\ &((x = y) \vee Edge(x, y) \vee Edge(y, x))] \\ &\wedge Sum\{w : Weight(v, w)|Clique(v)\} \geq K\} \end{aligned}$$

If we want to solve a different problem on node-weighted graphs, we don't need to change the grammar, and may not need to change the vocabulary map. To solve the Graph Colouring problem on the same set of graphs files, we would use the same grammar, but we would remove the vocabulary map rule corresponding to the $Node$ grammar rule, because we don't care about the node weights. We can also add a rule to fix a set of colours. Then the specification formula given in Example 1 can be used to find proper colourings.

The grammar for DIMACS graph format can be used to solve many types of graph problems by providing a vocabulary map and specification for each problem. This is possible because the grammar is dependant on the format of the input files but not on the problem to be solved.

B. Propositional Logics

Our grammar for propositional logic is given in Example 3. For a vocabulary map that maps formulas to structures as described in Section IV-B, the following first order formula constitutes a specification of standard propositional logic satisfiability.

$$\begin{aligned} &\forall f, f_1, f_2[And(f, f_1, f_2) \supset \\ &\qquad (True(f) \equiv (True(f_1) \wedge True(f_2)))] \\ &\wedge \forall f, f_1, f_2[Or(f, f_1, f_2) \supset \\ &\qquad (True(f) \equiv (True(f_1) \vee True(f_2)))] \\ &\wedge \forall f, f'[Not(f, f') \supset (True(f) \equiv \neg True(f'))] \\ &\wedge True(F) \end{aligned}$$

Many semantics are possible for the same set of formulas described by our grammar for propositional logic. The semantics are given by the specification, so we can interpret the same formulas differently by changing the specification. In particular, the following specification defines a standard V-valued propositional logic, for any positive integer V.

$$\begin{aligned} &\forall f \exists v[Value(f, v)] \\ &\wedge \forall f \forall v[Value(f, v) \supset \neg(\exists v' < v(Value(f, v')))] \\ &\wedge \forall f, f_1, f_2[And(f, f_1, f_2) \supset (\exists v \forall v_1, v_2(Value(f, v) \\ &\qquad \wedge Value(f_1, v_1) \wedge Value(f_2, v_2) \\ &\qquad\qquad \wedge (v = Min(v_1, v_2))))] \\ &\wedge \forall f, f_1, f_2[Or(f, f_1, f_2) \supset (\exists v \forall v_1, v_2(Value(f, v) \\ &\qquad \wedge Value(f_1, v_1) \wedge Value(f_2, v_2) \\ &\qquad\qquad \wedge (v = Max(v_1, v_2))))] \\ &\wedge \forall f, f'[Not(f, f') \supset (\exists v \forall v'(Value(f, v) \\ &\qquad \wedge Value(f', v') \wedge (v = (V - v'))))] \\ &\wedge Value(F, V) \end{aligned}$$

It is possible to represent formulas in different logics using the same grammar and vocabulary map because the grammar and map only define which features are semantically meaningful, not how those semantics are defined.

C. Integer Difference Logic (a.k.a. Separation Logic)

Syntactically, Difference Logic is essentially propositional logic except that atoms are arithmetic expressions of the form $x - yOPc$, where x and y are (integer valued) variables, c is an integer constant, and OP is one of $=,<,>$. For example

$$(((x - y < 9) \vee (x - y = 9)) \wedge (\neg((y - z > 6)))).$$

We can produce a grammar for this logic by modifying our grammar for propositional logic, replacing the Atom rule of Example 3 with the following rules for the more complex atoms:

$$\begin{aligned} Atom &\rightarrow \text{“(”}\ LessThan\ \text{“)”}|\text{“(”}\ Equal\ \text{“)”}| \\ &\qquad \text{“(”}\ GreaterThan\ \text{“)”} \\ LessThan &\rightarrow Variable\text{“}-\text{”}Variable\text{“}<\text{”}Constant \\ Equal &\rightarrow Variable\text{“}-\text{”}Variable\text{“}=\text{”}Constant \\ GreaterThan &\rightarrow Variable\text{“}-\text{”}Variable\text{“}>\text{”}Constant \\ Variable &\rightarrow \_character[lower] \\ Constant &\rightarrow \_integer[IntMAX : IntMIN] \end{aligned}$$

Given a similar extension of the vocabulary map, the following formula is a specification for integer difference logic satisfiability.

$$\begin{aligned}
&\forall v \exists n[Value(v,n)] \\
&\wedge \forall v \forall n[Value(v,n) \supset \neg \exists n' < n(Value(v,n'))] \\
&\wedge \forall f, f_1, f_2[And(f,f_1,f_2) \supset \\
&\qquad (True(f) \equiv (True(f_1) \wedge True(f_2)))] \\
&\wedge \forall f, f_1, f_2[Or(f,f_1,f_2) \supset \\
&\qquad (True(f) \equiv (True(f_1) \vee True(f_2)))] \\
&\wedge \forall f, f'[Not(f,f') \supset (True(f) \equiv \neg True(f'))] \\
&\wedge \forall f, v_1, v_2, n_1, n_2, c[LessThan(f,v_1,v_2,c) \supset \\
&\qquad (True(f) \equiv Value(v_1,n_1) \wedge \\
&\qquad Value(v_2,n_2) \wedge (n_1 - n_2 < c))] \\
&\wedge \forall f, v_1, v_2, n_1, n_2, c[Equal(f,v_1,v_2,c) \supset \\
&\qquad (True(f) \equiv Value(v_1,n_1) \wedge \\
&\qquad Value(v_2,n_2) \wedge (n_1 - n_2 = c))] \\
&\wedge \forall f, v_1, v_2, n_1, n_2, c[GreaterThan(f,v_1,v_2,c) \supset \\
&\qquad (True(f) \equiv Value(v_1,n_1) \wedge \\
&\qquad Value(v_2,n_2) \wedge (n_1 - n_2 > c))] \\
&\wedge True(F)
\end{aligned}$$

VI. A Prototype System

We have implemented a prototype system in Python. For mapping the input string to a structure, it requires two specification files, a grammar file and a vocabulary map file.

A. Parsing

The prototype uses NLTK: the Natural Language ToolKit [13], a natural language processing toolkit for Python, as the parser for instance strings. The form of the grammars is a natural ASCII version of that used in our examples.

The parser constructs a parse tree where each node of the tree corresponds to an application of a grammar rule. This parse tree contains nodes which do not correspond to any semantically meaningful feature. For example, the parse tree for a formula of propositional logic contains many $Formula$ nodes, but these serve no semantic purpose, so we remove these nodes. The vocabulary map tells the system which kinds of nodes are needed, and which can be removed.

B. Mapping Parse Tree to Structure

The vocabulary map file specifies how to map the parse tree to an abstract structure. In particular, it fixes the vocabulary of the instance structure, and identifies which kind of node in the parse tree corresponds to which vocabulary symbol. A predicate tag, of the form <predicate>, is used for this. The vocabulary symbol is given by the name attribute, and the corresponding node type is given by a grammar tag. For example, the following predicate tag is used for the connective $\wedge$ in propositional logic.

Example 7: The predicate tag

```
<predicate name="And">
    <grammar>And</grammar>
    <type>number_children</type>
</predicate>
```

The type tag specifies how the arity of the corresponding relation, and the contents of its tuples, are determined. In the example for $\wedge$, keyword `number_children` indicates that the first column (argument) is the identifier of the node representing the element and the remaining columns (arguments) are the identifiers of its children.

Terminal nodes are identified by the terminal tag, <terminal/>, appearing in the predicate tag.

Example 8: The predicate tag

```
<predicate name="Atom">
   <grammar>Atom</grammar>
   <type>number</type>
   <terminal/>
</predicate>
```

represents the atoms of a formula.

The table identifying the atoms of the formula has only one column which is the identifier of the node representing the element that is an atom. This is indicated in the type tag using the keyword `number`.

VII. Mapping Structures to Solver Instances

In the prototype system, the mapping of an instance structure to an instance file for a specific solver is determined by at solver format file. We will illustrate this using our running example, formula F, and the IDP System.

Example 9: Assume we have an IDP System specification for semantics of propositional logic equivalent to our specification in Section V. The IDP System instance file for our formula F is:

```
  structure Formula:Propositional {
  Formula = {1..5}
  And = {(1,2,5)}
  Or = {(2,3,4)}
  Not = {(5,3)}
  Atom = {(3);(4)}
}
```

The line `Formula = {1..5}` gives the list of domain elements corresponding to sub-formulas, and the following lines give the relations of the structure as lists of tuples. These correspond directly to the abstract structure, as described in Example 6.

The solver format file must specify how to generate this presentation of the structure.

Example 10: Here is a solver format file for the IDP System.

```
<idp>
   <structure>
      structure $name$:$vocabulary$ {\n
      $domains$$relations$}\n
   </structure>
   <domain>
      $name$ = { $lower$..$upper$ }\n
   </domain>
   <relation separator=";">
```

```
      $name$ = { $tuples$ }\n
   </relation>
</idp>
```

The <structure> tag describes the overall format: the structure name and vocabulary name (which come from the vocabulary map), followed by (enclosed in braces) the descriptions of the domains and then the relations, as indicated by the order of $domains$ and $relations$. The <domain> tag gives the format of an IDP System domain description (of the sort we need here), which consists of the name followed by an appropriately formatted list of elements, in this case those in the range between $lower$ and $upper$. The <relation> tag specifies the format of descriptions of relations. In this case, it is the name followed by an appropriately formatted list of tuples.

VIII. Discussion

Systems that allow users to solve combinatorial problems, including optimization problems and problems that arise in software and hardware design and verification, are becoming more powerful, and more practical. For the most part, users can apply these by writing high-level declarative specifications, rather than writing code. We have described here a method and a prototype tool that addresses the remaining non-declarative aspect of using these tools: mapping problem instances from their application-dependant native format to the instance (or data) format of a particular solver. The tool is fairly general, and can handle a wide range of instance description languages. As a result, it can be used to declaratively turn model-and-solve systems into special purpose solvers, for example model finders for a variety of logics, in very little time.

Representation of syntactic objects in structures has been used at least since the 1970s in meta-programming schemes, such as that in the Prolog programming language. It is used in the area of linguistics known as model theoretic syntax [14]. The first use we are aware of in model-expansion based problem solving is in [9], where propositional logic, constraint satisfaction problems, and answer set programs are represented as structures. The IDP system for knowledge representation and reasoning has a very powerful "bootstrapping" facility [15], which operates by transforming formulas in its representation language to structures, and operating on those structures. It can also be used to produce "instant model finders" for many logics, provided they are syntactically fragments of the IDP system language, but the facility is primarily a tool for IDP system developers. Our tool is intended as a front-end, for users of a variety of solving systems, and, as far as we understand, is more flexible regarding input syntax.

Our tool and method both have many limitations at this point. The parser currently used imposes inconvenient restrictions and causes major performance problems, presumably because it was not designed to parse the sort of inputs we face, such as large CNF formulas or graphs. For serious use, we need to extend the tool design with a proper type system. Currently, the method for mapping structures to solver instances does not have the generality that might be desired.

Nonetheless, our experience in designing and building the tool suggests the approach is potentially very useful. In particular, as we move from tools for combinatorial problem solving being only for specialists to being usable by a wide range of workers, and from only a few examples of web-based public access to such tools to serious high-performance cloud-based services, a tool such as this should be a standard part of the cloud-based service.

References

[1] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[2] T. Koch, "Rapid mathematical prototyping," Ph.D. dissertation, Technische Universität Berlin, 2004.

[3] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard cp modelling language," in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, ser. CP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 529–543. [Online]. Available: http://dl.acm.org/citation.cfm?id=1771668.1771709

[4] J. Wittocx, M. Marien, and M. Denecker, "The IDP system: a model expansion system for an extension of classical logic," in *Proc., LaSh 2008*, 2008.

[5] A. Aavani, X. N. Wu, S. Tasharrofi, E. Ternovska, and D. G. Mitchell, "Enfragmo: A system for modelling and solving search problems with logic," in *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, ser. Lecture Notes in Computer Science, N. Bjørner and A. Voronkov, Eds., vol. 7180. Springer, 2012, pp. 15–22. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28717-6

[6] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

[7] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," *ACM Trans. Comput. Log.*, vol. 7, no. 3, pp. 499–562, 2006.

[8] H. B. Enderton, *A mathematical introduction to logic*. Academic Press, 1972.

[9] D. G. Mitchell and E. Ternovska, "A framework for representing and solving NP search problems," in *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press / The MIT Press, 2005, pp. 430–435. [Online]. Available: http://www.aaai.org/Library/AAAI/2005/aaai05-068.php

[10] ——, "Expressiveness and abstraction in ESSENCE," *Constraints*, vol. 13(2), pp. 343–384, 2008.

[11] S. Tasharrofi, X. N. Wu, and E. Ternovska, "Solving modular model expansion: Case studies," in *Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, and A. Wolf, Eds., vol. 7773. Springer, 2011, pp. 215–236.

[12] D. S. Johnson and M. A. Trick, *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. American Mathematical Soc., 1996, vol. 26.

[13] E. Loper and S. Bird, "NLTK: the natural language toolkit," *CoRR*, vol. cs.CL/0205028, 2002. [Online]. Available: http://arxiv.org/abs/cs.CL/0205028

[14] J. Rogers, "A model-theoretic framework for theories of syntax," in *34th Annual Meeting of the Association for Computational Linguistics, 24-27 June 1996, University of California, Santa Cruz, California, USA, Proceedings.*, A. K. Joshi and M. Palmer, Eds. Morgan Kaufmann Publishers / ACL, 1996, pp. 10–16. [Online]. Available: http://aclweb.org/anthology-new/P/P96/

[15] B. Bogaerts, J. Jansen, B. De Cat, G. Janssens, M. Bruynooghe, and M. Denecker, "Meta-level representations in the IDP knowledge base system: Towards bootstrapping inference engine development," 2014, 2014 Workshop on Logic and Search (LaSh '14).