Instance Transformation for Declarative Solvers

by

Megan O'Connor

B.Sc. [Hons.], Memorial University of Newfoundland, 2013

Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2015

Approval

Name:	Megan O'Connor
Degree:	Master of Science (Computing Science)
Title:	*Instance Transformation for Declarative Solvers*
Examining Committee:	**Gregory Baker** (chair) Senior Lecturer

David Mitchell
Senior Supervisor
Associate Professor _____

Eugenia Ternovska
Supervisor
Associate Professor _____

William N. Sumner
External Examiner
Assistant Professor _____

Date Defended: December 4, 2015

Abstract

Declarative specification-based problem solving systems, or "model-and-solve" systems, solve combinatorial search problems based on specifications in high-level declarative languages. Users of these systems can solve challenging combinatorial problems by describing what a solution is, rather than describing an algorithm for finding one. While the problem specifications are declarative, users of existing systems must write programs to transform problem instances into solver-specific formats, so problem solving is not fully declarative. We describe a purely declarative method for transforming instances from native file formats to solver-specific formats. We also describe a prototype implementation which, used together with existing declarative solvers, provides fully declarative problem solving. The method can also be seen as a way to produce model finders for new logics, of moderate expressive power, purely declaratively. We illustrate application of the method to a variety of problems, including graph problems and logical satisfiability problems.

Keywords: declarative problem solving;knowledge representation and reasoning;constraint modelling languages;mathematical programming;answer set programming

Dedication

To my grandfather,
Denis Anthony O'Connor

Acknowledgements

I would like to thank my senior supervisor David Mitchell for introducing this line of research and for supporting me throughout my degree. I also want to thank my sisters and grandmother for all of their encouragement. Finally, I would like to thank my parents for all the support and encouragement they have given me throughout my education.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

A declarative problem-solving system (or model-and-solve system) is a software system where users provide a problem specification which describes a solution for an instance of a problem in terms of the problem domain. This differs from other problem-solving methods which require the user to describe a process, or algorithm, to find a solution for an instance. Our interest is in declarative solvers for combinatorial search and optimization problems. Declarative solving systems are used in mathematical modelling, constrained optimization, software design and many other areas.

Use of this kind of system is conceptually simple, as illustrated in Figure 1.1. 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 there is one, or reports there is none. For example, for the problem of Graph Colouring, the problem specification says (in syntax of the relevant language) that, in a proper colouring, 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 many of the languages used by declarative problem-solving systems. Then, the specification and a particular graph are given as input to the system, which will try to construct a proper colouring.

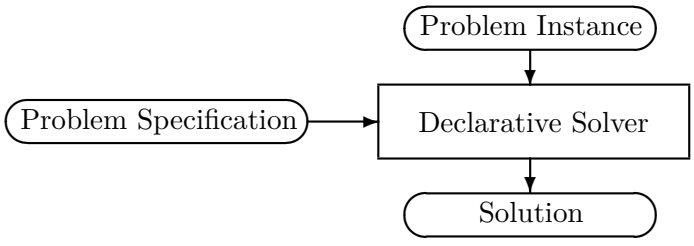


Figure 1.1: Declarative Problem Solving Systems

Implemented systems for solving combinatorial problems based on declarative specifications are growing in variety, number, and range of effective application. Examples of declarative problem solving systems include systems using the following specification languages: the Alloy software modelling language [8], algebraic modelling languages used in mathematical programming such as ZIMPL [11], constraint modelling languages used in combinatorial optimization such as MiniZinc [16], and logic-based knowledge representation languages such as used by the IDP System [23], Enfragmo [1], and Answer Set Programming systems such as `clingo` [7] and DLV [12]. We briefly describe several of these systems in Chapter 2.

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. Because they are used without writing executable programs, they are sometimes called "model-and-solve" systems.

These systems are currently used primarily by specialists as a more efficient way to tackle challenging combinatorial search and optimization problems than developing new algorithms and implementations for each problem. 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 and allows them to apply high-performance solving technology to their own problems. As the quality of these systems improve, they become more accessible and more likely to be successfully used by such workers. Moreover, 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.

However, the above description of using such a system left out an essential step: the problem instance must be put into the input format of the solver. The problem instances for the various problems users might want to solve all have distinct native file formats, which can vary considerably. To use a declarative solver a user must transform these problem instances into the file format of the solver. For example, suppose we are using the IDP System [23] to solve a particular application problem, and there is a collection of benchmark instances in the format of another solver. We need to translate the instances from their original file format into the instance file format of the IDP System. This requires writing a program in a traditional programming language. As a consequence, the solving is not purely declarative, and these "declarative" systems cannot be used by non-programmers.

1.2 Contribution

The purpose of the work reported here is to introduce a method, and a prototype system called INSTLATOR, that addresses this problem. While it seems logical to directly translate instance strings to a specific solver instance format this would require a distinct translator for each target solver. Instead we would like a method that easily supports translation to the various solver formats without making stand-alone translators for each one. Our method is based on the logical formalization of combinatorial problem solving, introduced in [15], which allows us to abstract away from the particular syntax of any given solver or instance description language. In this formalization, a problem instance is simply a finite structure, in the sense the term "structure" is used in mathematical logic. Our method involves two main stages. The first maps a string in an instance description language to a structure, and the second maps the structure to an instance description in the specific format used by a particular model-and-solve system. The system, 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. The separate stages of the system makes translating an instance string to multiple target solvers much easier.

The use of separate stages in our method results in an instance structure being produced which is independent of any particular solver. A representation of these instance structures can be stored and later translated to multiple target solver formats. This provides a way to create a solver-independent database of benchmark problem instances which could be quite valuable to the declarative problem solving communities. For example, suppose we need to solve many challenging instances of an application problem, and want to test the performance of various solvers. Writing a program to transform the instances for their original file format to the instance formats of the various solvers would be tedious. With our method, we can generate a database of the instances in generic form, and then use the tool again to transform them to the format of each solver we wish to evaluate. Currently there is very little comparison of tools and methods between the communities, partly due to them using different sets of benchmark problems and having different instance formats.

As we will see, our solution to the instance format problem will allow us to do something much more general as well. Researchers and developers frequently find that there is a use for some logic or declarative language for which they may not have software tools, such as model finders or theorem provers, available. This may be because they obtain expressions in such a language from other sources, or because they devise a new language which seems of potential use. Building a model finder can involve serious software development effort. Using our method, one can turn a general purpose declarative solver, such as an Integer Programming solver, into a model finder for a wide variety of logics with only a page or so of declarations, and no executable code.

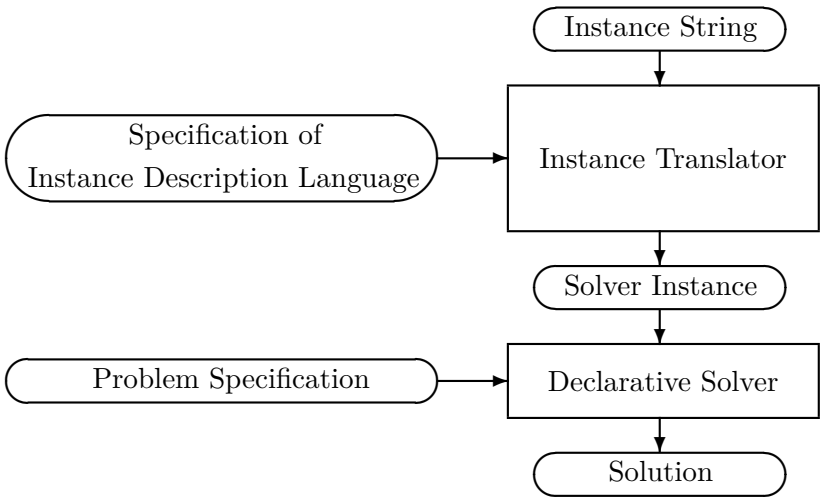


Figure 1.2: Instance Translation as a Front End for a General Purpose Declarative Solver.

1.3 Related Work

We are aware of no other work directly addressing the same problem we are. The problem is one of language translation, and thus related to compiler technology. We must still investigate whether compilation technology, such as attribute grammars, might be fruitfully used for our task. The scheme used by our method, as described in Chapter 3, is similar to source-to-source translation [18]. However, the semantics of our languages are quite different from those of programming languages.

Central to our method is the representation of a syntactic object as a structure. This has been used in the area of linguistics known as model theoretic syntax [19]. It has been used implicitly in meta-programming schemes, such as that in the Prolog programming language. Meta-programming is the ability to treat computer programs as data within another program. Prolog has many predicates where Prolog terms are interpreted as Prolog clauses and goals. These predicates give Prolog a meta-programming (or meta-logical) facility which allow clauses and goals to be variables within a program [17].

The first use of structures representing syntactic objects we are aware of in the formalization of model-and-solve systems is in [15], 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 [3], which operates by transforming formulas in its problem representation language into structures, and thus changing formula manipulation tasks into declarative problem solving on structures. Like our system, it can be used to declaratively create model finders for other logics, but only if those logics are syntactic fragments of the IDP system language. The facility is primarily a tool for IDP system developers. Our system is intended

as a front-end, for users of a variety of solving systems, and, as far as we know, is more flexible regarding input syntax.

Software such as spreadsheet and database systems have facilities for manipulating input formats. Also, software such as document and image processors have format translation abilities. These operate on very specific applications which are far-removed from declarative problem solving systems and we have not investigated whether the techniques used in these applications might be relevant to ours or not.

1.4 Organization of the Document

In Chapter 2 we give some formal preliminaries and introduce declarative problem solving and model expansion. We explain the concept of model expansion and introduce a number of declarative problem solving systems. We give an overview of our method and introduce the INSTLATOR system in Chapter 3, with further details on the two stages of our method in Chapters 4 and 5. Chapter 4 provides details on the first stage of our method which maps instance strings to instance structures. We describe the two steps, parsing and mapping, of this stage and introduce the declarative descriptions used in this stage of our method. Chapter 5 provides details on the second stage of our method which maps instance structures to solver-specific instances. We also introduce the XML representation used by the INSTLATOR system to store instance structures and describe the formatting step performed in this stage of the method as well as the declarative description used in this stage. Finally, we provide our conclusion in Chapter 6.

Chapter 2

Declarative Problem Solvers

Developing a method for our task with a reasonable level of generality requires a suitable abstraction of what a problem instance is. In this section, we describe a logical formalization of declarative specification-based problem solving that abstracts away from most system-specific and problem-specific details, thus providing such an abstraction. While superficially very different, all of the specification-based solvers described in Section 2.3, and many others, can be seen to solve the same general abstract problem and thus formalized in this 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.

2.1 Preliminaries

We assume the reader is familiar with the basics of propositional and first-order logics. Our propositional formulas have atoms P_i, connectives $\wedge$, $\vee$, $\neg$ and parentheses $(,)$, and are defined by the usual inductive construction. We use the standard truth-functional semantics defined by the satisfaction relation between formulas and truth assignments to the propositional atoms.

First order formulas have predicate symbols P_i, functions symbols f_i, variable symbols x_i and quantifier symbols $\forall$ and $\exists$, as well as connectives $\wedge$, $\vee$, $\neg$ and parentheses $(,)$. Formulas are defined by the usual inductive construction. 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 $\mathcal{A}$. Formulas are defined by the usual inductive construction, with

the standard truth-function semantics defined by the satisfaction relation between formulas and truth assignments of the predicate and functions of a τ-structure. A good reference for details is [5].

2.2 A Logical Formalization of Model-and-Solve Systems

The declarative solving systems we are interested in all solve essentially the same task, but their specification languages, instance file formats, and associated terminology for declarative problem-solving tools can differ considerably. Our concern here is with the syntax of instance (or data) descriptions used by these systems. For our method to have a reasonable degree of generality, we need an abstract notion of a problem instance that is not tied to any particular solver syntax.

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 weights. The solution to be output is a related collection of functions and relations. For example, a subset of the edges of a given graph (e.g., a minimum-weight spanning tree or matching), or a function from vertices to colours. For simplicity of presentation, in the remainder we often assume we have relations only, and no functions. (A k-ary function f may be viewed as the $k+1$-ary relation known as its graph, namely $R_f = \{\vec{x}, y \mid f(\vec{x} = y)\}$.) The problem specification describes, in some formal language, the relationship between the input relations, which correspond to the problem instance, and the output relations, which correspond to the problem solution. 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, $\mathcal{A}$ is a σ-structure, and $\mathcal{B}$ is a τ-structure. If B has the same universe as A (that is, $A = B$), and has the same interpretation for all relation symbols of 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 $\tau = \sigma \cup \gamma$ be the combined instance and solution vocabulary. If σ-structure $\mathcal{A}$ is a problem instance, then any structure B that is an expansion of A to τ, is a structure consisting of an instance together with a "possible solution". In this formalization, every element of the

solution is also an element of the instance. The formalization can be extended to the more general setting where solutions contain elements which are not part of the instance, but doing so adds complexity which is not needed for our purposes.

With instances and corresponding solutions viewed this way, a problem 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 tuple containing the symbols of τ. Thus, $\mathcal{A}$ has a solution if it is a model of $\exists \vec{R} S$, and a solution for $\mathcal{A}$ is an expansion of $\mathcal{A}$ that satisfies the formula S. Hence [15] used the term "Model Expansion" for the underlying formal task of specification-based solvers.

Example 2.1. 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 relation symbol that 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:

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

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

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

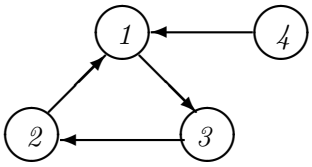
$$\begin{aligned} &\forall x \exists c[Col(x,c)] \\ \wedge &\forall x \forall c[Col(x,c) \supset \neg \exists k((c \neq k) \wedge Col(x,k))] \\ \wedge &\forall x \forall y[E(x,y) \supset \forall c(\neg(Col(x,c) \wedge Col(y,c)))] \end{aligned}$$

The first constraint states that every vertex is assigned a colour, the second constraint states that a vertex can be assigned at most one colour and the third constraint states that adjacent vertices cannot be assigned the same colour.

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 [14] and [21]. So, this formalization is indeed applicable to real systems.

2.3 Examples of Declarative Problem-Solving Systems

There is a growing number and variety of declarative problem solving systems. Here we introduce some of these systems and show how each represents the same problem. Consider the following graph G.



For each declarative solver described below we will show graph G represented in the solver's instance format. We will also write the following constraint in the specification (or modelling) language of each solver.

$$\forall x \forall y [E(x,y) \supset \forall c(\neg(Col(x,c) \wedge Col(y,c)))] \tag{2.1}$$

The constraint is from the Graph Colouring problem and states that if there is an edge between two vertices then the vertices cannot be assigned the same colour. For some of the declarative solvers Col will be a node colouring function and for the others Col will be a relation.

The representations shown in the following sections demonstrate how the instance file formats of declarative solver can vary depending on the intended application problem, target user community, or developer community.

2.3.1 ZIMPL

ZIMPL (Zuse Institute Mathematical Programming Language) is a declarative problem-solving system that translates a mathematical model of a problem into a linear or nonlinear integer mathematical program expressed in .lp or .mps file format, which can be read and solved by many LP or MIP solvers [11].

The representation of graph G in ZIMPL requires two text files. The following is the first file which provides the set of vertices.

```
# Set V
1
2
3
4
```

The other file provides the set of edges, as follows.

```
# Set E
1 3
2 1
3 2
4 1
```

The following is the representation of our constraint 2.1 for ZIMPL.

```
constraint forall <x>  in V do forall <y> in V do
 if ( <x,y> in E ) then col[x] ~= col[y] end;
```

2.3.2 MiniZinc

MiniZinc is a constraint modelling language for specifying constrained optimization and decision problems over integers and real numbers. It uses a somewhat C-like syntax to describe constraints involving quantification, a wide variety of arithmetic constraints, and logical connectives [16].

The following is our graph G represented in the format for MiniZinc instances. In addition to the set of edges, provided as an array, we must also provide the size of sets V, E, and C, the sets of vertices, edges and colours respectively.

```
sizeV = 4;
sizeE = 4;
sizeC = 3;
E = [ | 1, 3 | 2, 1 | 3, 2 | 4, 1 | ];
```

The following is the representation of our constraint 2.1 for MiniZinc.

```
constraint forall ( x in V, y in V, z in 1..sizeE ) (
 ( E[z,1] = x /\ E[z,2] = y ) => ( Col[x] != Col[y] ) ) );
```

Complete details of a MiniZinc implementation of graph colouring are available from the MiniZinc website [16].

2.3.3 IDP System

The IDP system is a knowledge base system using a language which is an extension of first order logic with inductive definitions, aggregates and types, called FO(·) by the system authors. A knowledge base system is a system that supports multiple forms of inferences for the same knowledge base [23].

The following is our graph G in the IDP System's instance format. In addition to the sets of vertices and edges we must also provide the set of colours `C`.

```
structure Graph:Colouring {
 V = {1..4}
 C = {1..3}
 E = {(1,3);(2,1);(3,2);(4,1)}
}
```

The following is the representation of our constraint 2.1 in the language of the IDP System.

```
! x [V] y [V] : E(x,y) => Col(x) ~= Col(y)
```

Here `!` represents the quantifier ∀.

2.3.4 Enfragmo

Enfragmo [1] is a solver explicitly based on model expansion, with a specification language that extends first order logic with types, arithmetic and aggregates, similar to that of the IDP system.

This is the representation of our graph G in Enfragmo's instance format.

```
TYPE V [1..4]
TYPE C [1..3]
PREDICATE E
 (1,3)
 (2,1)
 (3,2)
 (4,1)
```

This is our constraint 2.1 represented for Enfragmo.

```
! x : V y : V c : C : ( E(x,y) => ( ~ ( Col(x,c) & Col(y,c) ) ) );
```

Complete details of the Enfragmo implementation of graph colouring are available from the Enfragmo User Manual [20].

2.3.5 clingo

Answer Set Programming is a declarative problem solving framework in which problem specifications are described using logic programs with Stable Models (or Answer Set) semantics. The "Potsdam Answer Set Solving Collection" (Potassco) gathers a variety of tools for Answer Set Programming, including grounder `gringo`, solver `clasp`, and their combination within the integrated ASP system `clingo` [7].

The following is our graph G represented in the instance format for `clingo`. The set V is represented by predicate `node` and set E by predicate `edge`.

```
node(1..4).
edge(1,3). edge(2,1). edge(3,2). edge(4,1).
```

The following is a `clingo` representation of our constraint 2.1 from the solution for Graph Colouring.

```
:- edge(X,Y), color(X,C), color(Y,C).
```

Complete details of the `clingo` implementation of graph colouring are available from the Potassco User Guide [6].

2.3.6 Alloy

Alloy is a lightweight modelling language for software design. It is amenable to a fully automatic analysis, using the Alloy Analyzer, and provides a visualizer for making sense of solutions and counterexamples it finds. The language is a simple but expressive logic based on relations and the syntax is designed for building models incrementally [8]. The model finding engine used by Alloy is a constraint solver that implements model expansion for relational first order logic.

A full implementation of the Graph Colouring problem is available from the Alloy website [22]. The implementation supports three graph formats, one of which is the DIMACS graph format [10]. The following is our graph G represented in DIMACS graph format. (In Chapter 3 we describe the application of our method to DIMACS graph format.)

```
p edge 4 4
e 1 3
e 2 1
e 3 2
e 4 1
```

The following is the code which corresponds to our constraint 2.1 in the Graph Colouring implementation.

```
for(int i = 0; i < vcolors.length; i++) {
 final int[] neighbors = graph[i];
 final int max = neighbors.length;
 final Relation vcolor = vcolors[i];
 for(int j = 0; j < max; j++) {
  formulas.add( vcolor.intersection(vcolors[neighbors[j]]).no() );
}
```

Complete details of the implementation of graph colouring are available from the Alloy website [22].

Chapter 3

Method Overview

The purpose of the method is to map a string, which describes an instance of a problem, into the instance (or data) format for a chosen declarative problem solving tool. Given the formalization of search problems as model expansion, described in Chapter 2, we may accomplish our translation of instance description strings to solver input formats in two stages, as illustrated in Figure 3.1. The first stage maps an instance string to an abstract instance representation. The second stage maps an abstract instance representation to a string describing that structure in the format of a particular solver. The first stage itself is carried out in two steps. The first step parses the instance string based on the syntactic features of the language in which the problem instances are described. Any reasonable language will have syntactic constructs corresponding to the semantically meaningful properties of the problem instance. The second step maps the semantically significant syntactic features to the semantic vocabulary of an abstract instance representation. The solver-specific instance produced by our method can be used, together with a problem specification written in terms of the appropriate vocabulary, by the declarative solver to find a solution.

Each stage of the method, as illustrated in Figure 3.1, requires declarative descriptions of the steps performed by the stage. These declarative descriptions are provided by the following three 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 defines the vocabulary of the instance structure and 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.

The first stage of our method requires a grammar and a vocabulary map to preform the mapping from an instance string to an instance structure. The grammar is used during the

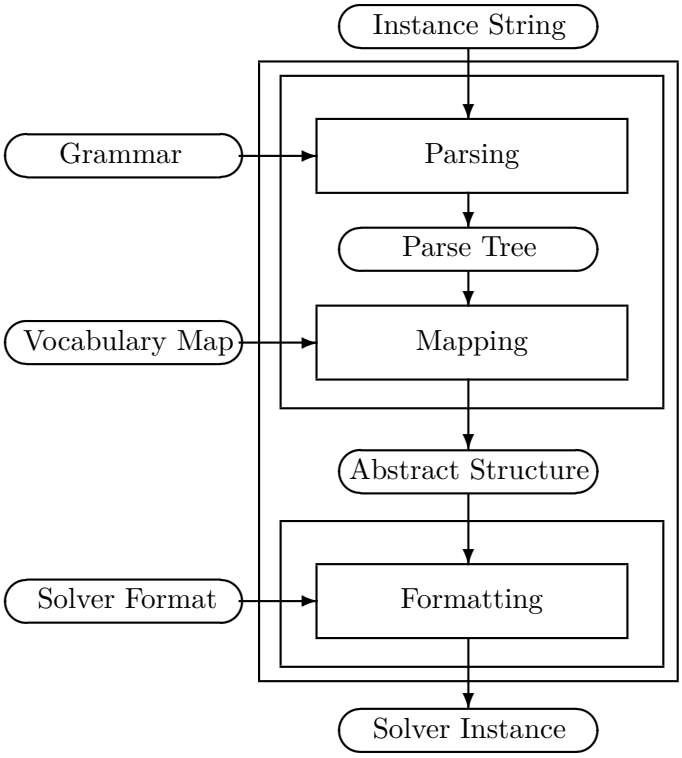


Figure 3.1: Internal scheme of the instance translator.

first step to identify the syntactic features of the language in which the instance is described. The vocabulary map is used during the second step to identify the semantically significant syntactic features of the instance string and is also used to map these syntactic features to the semantic vocabulary of the instance structure. The second stage of our method requires a solver format description to perform the mapping from an instance structure to a solver-specific instance file. This mapping is not dependent on the grammar or the vocabulary map and therefore the first and second stages of the method can be used independently. For example, given an instance of some problem, we may generate a corresponding instance structure and store it without regard to which solver or solvers we may want to apply to it. Later, the abstract instance structure can be used to generate instance files in the formats of one or more solvers. This means that transformation to the formats of new solvers can be added without needing to modify the transformation from instance strings to structures. It also offers the possibility to store a collection of benchmark instances in a solver independent form for later use.

In the following two chapters, we will describe the two stages in further detail. We will illustrate the method using propositional logic as an example problem. That is, we suppose that we want to use a general-purpose declarative solver for finding satisfying assignments to formulas of propositional logic.

3.1 The INSTLATOR System

The INSTLATOR system is an implementation of our method as illustrated in Figure 3.1. The system consists of two tools, each of which corresponds to a stage of our method. The first tool, an instance structure generator as depicted in Figure 3.2, implements the method for mapping of instance strings to instance structures described in detail in Chapter 4. The tool stores a representation of the instance structure in the form of an XML file. The second tool, a solver instance generator as depicted in Figure 3.3, implements the method for mapping instance structures to solver instances described in Chapter 5. This tool formats a stored representation of an instance structure to a solver-specific instance file. A complete description of the XML representation of instance structures is given in Chapter 5. Both tools are Python programs which require Anaconda 2.1.0 or higher to be installed. A free distribution of Anaconda is available from Continuum Analytics [2].

3.1.1 Instance Structure Generator

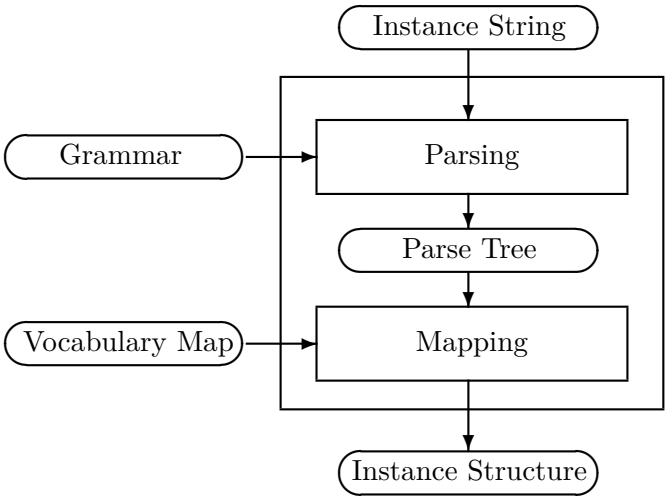


Figure 3.2: Internal scheme of the instance structure generator.

This tool parses instance strings and maps them to instance structures as described in Chapter 4. The instance structure generator uses the parser included in NLTK: the Natural Language ToolKit [13], a natural language processing toolkit for Python, as the parser for instance strings. To use the tool navigate to the directory containing the INSTLATOR system and run the following command.

```
$ py generate_instance_structure.py instance.txt grammar.txt
vocabulary_map.xml > instance_structure.xml
```

The command uses four files

1. The file `instance.txt` contains the instance string. It is not restricted to have a specific file extension.

2. The file `grammar.txt` contains the grammar used to parse the instance string.

3. The file `vocabulary_map.xml` contains the vocabulary map used to map the instance string to a generic instance.

4. The file `instance_structure.xml` will store the instance structure generated by the tool.

3.1.2 Solver Instance Generator

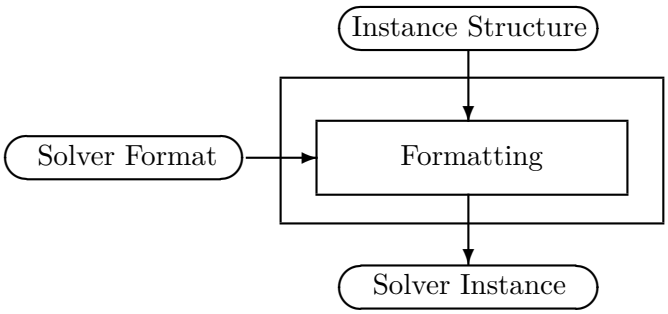


Figure 3.3: Internal scheme of the solver instance generator.

This tool maps instance structures to solver instances as described in Chapter 5. To use the tool navigate to the directory containing the INSTLATOR system and run the following command.

```
$ py generate_solver_instance.py instance_structure.xml
solver_format.xml > solver_instance.txt
```

The command uses three files

1. The file `instance_structure.xml` contains the instance structure.

2. The file `solver_format.xml` contains the solver format description for the declarative solver.

3. The file `solver_instance.txt` will store the solver instance generated by the tool and should have the file extension require by the solver.

Chapter 4

Mapping Strings to Structures

In this chapter we describe the mapping from instance strings to instance structures. In Sections 4.1 and 4.2 we describe the two main steps, using a formula of propositional logic as a simple running example. In Sections 4.3 and 4.4 we present a number of examples using graphs and extensions of propositional logic, which illustrate a number of further details. In Section 4.5 we show how problem instances for the same problem, but in different solver instance formats, can all be mapped to the same abstract structure.

4.1 Parsing Instance Strings

In the first step of this stage of our method, 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.

Example 4.1. Let F be the formula

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

For our example problem of propositional logic, here is a suitable grammar.

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

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 rules *And*, *Or* and *Not* give the form of sub-formulas involving the three connectives. The rule *Atom* indicates that an atom is a lower case character.

In Example 4.1, we used the notation preferred in typeset (or hand-written) work, but the actual strings used in practice are normally ASCII versions of these. Example 4.2 gives the ASCII version of the grammar, for the ASCII version of propositional logic formulas.

Example 4.2. The ASCII version of our formula F is

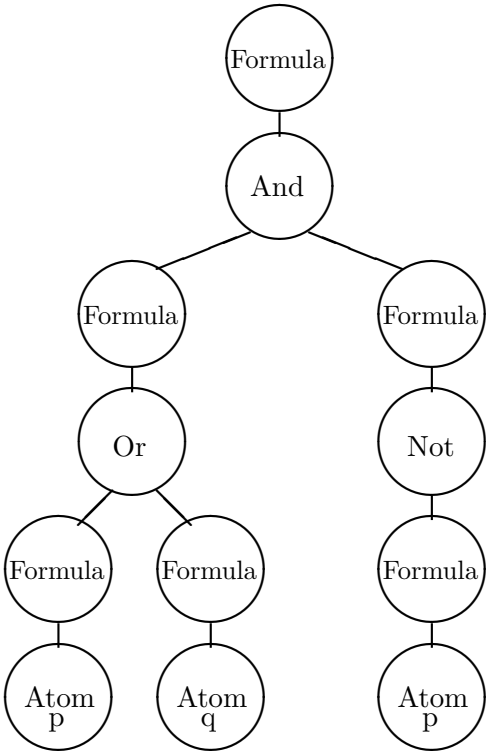
```
( ( p | q ) & ~ p )
```

and the ASCII version of our propositional logic grammar is

```
Formula -> Atom | "(" And ")" | "(" Or ")" | "(" Not ")"
And -> Formula "&" Formula
Or -> Formula "|" Formula
Not -> "~" Formula
Atom -> _character[lower]
```

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

Example 4.3. The parse tree P of formula F is



The parser constructs a parse tree P where each node of the tree corresponds to an application of a grammar rule. This parse tree contains nodes which do not carry any meaningful information about the corresponding formula. For example, the parse tree P for formula F produced by the parser will contain nodes of type `Formula` which indicates that the sub-string is a formula and is enclosed within parentheses. Nodes of this type do not carry any meaningful information about the formula. Nodes of type `And`, `Or`, `Not` and `Atom` already imply that a sub-string is a formula and parentheses do not have a semantic meaning within the formulas, except to uniquely determine the intended parsing. This means that nodes of type `Formula` are not semantically meaningful elements of propositional logic.

The vocabulary map identifies the nodes of the parse tree which will have corresponding semantic objects in the instance structure we construct from the tree. Therefore, it makes sense for the vocabulary map to also identify those nodes of the parse tree which should be ignored. Before proceeding to map the parse tree to the structure, the system removes these nodes. For propositional logic, our vocabulary will be

$$\tau_{PL} = [\text{Atom}, \text{Not}, \text{And}, \text{Or}]$$

where `Atom` is a unary predicate symbol, `Not` a binary predicate symbol, and `Or` and `And` are ternary predicate symbols. The relations denoted by these symbols each correspond to nodes with the same names in the parse tree. There is no vocabulary symbol corresponding to Formula nodes, so we will remove those.

In the vocabulary map, nodes of the parse tree which correspond to semantic objects in the instance structure are identified by a type tag of the form `<type>`. The instance structure vocabulary symbol is given by the name attribute. The grammar rules which correspond to the vocabulary symbol are identified in the grammar tag.

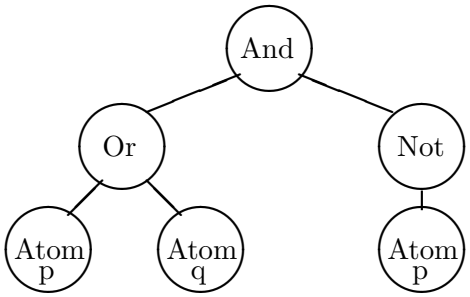
Example 4.4. The following type tag is used to specify the vocabulary symbol `Formula`.

```
<type name="Formula">
    <grammar>And,Or,Not,Atom</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
```

The grammar rules corresponding to the vocabulary symbol `Formula` are the rules *And*, *Or*, *Not* and *Atom*.

Any grammar rule which isn't assigned to a vocabulary symbol by the vocabulary map is ignored by the system. For convenience in constructing the abstract structure, the INSTLATOR system will remove any nodes corresponding to the ignored grammar rules by mapping the original parse tree to a tree containing only the semantically meaningful elements.

Example 4.5. After the nodes of type `Formula` are removed from the parse tree P of formula F, the new parse tree T contains only nodes of type `And`, `Or`, `Not` and `Atom`.



To view the complete syntax of grammar files see Appendix A.

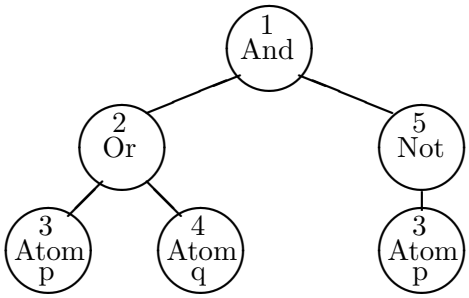
4.2 Structure Construction

In the second step, we want to map parse trees to instance structures. The relations in these structures should make the semantic content of the strings explicit.

For our 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.

During the mapping of the original parse tree P to the new tree T each node representing a semantically meaningful element is assigned an identifier. Identifiers are assigned based on the type of the semantic element specified by the vocabulary map. The vocabulary map provides a range for each type's identifiers using the range tag. Each range must have a lower bound, provided by the lower tag, which will usually be set to `1`. Nodes are assigned identifiers starting with the lower bound for the root of the parse tree T and assigned incrementally according to an in-order traversal of the parse tree. Leaves, which correspond to terminals in the grammar, are handled differently. The first leaf corresponding to any particular terminal is given the next node identifier, and any subsequent leaves with the same terminal symbol are given the same node identifier. The upper bound is determined by the system and will be set to the number of nodes assigned to each type. An upper bound can also set using an upper tag within the range tag, however this will override the value set by the system. In most cases, the system upper bound setting is appropriate

Example 4.6. The nodes of parse tree T with identifiers assigned.



Mapping the parse tree T 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.

The τ_{PL} structure $\mathcal{M}$ corresponding to our formula F has:

$$
\begin{aligned}
M &= \{1, 2, 3, 4, 5\} \\
Atom^{\mathcal{M}} &= \{(3), (4)\} \\
And^{\mathcal{M}} &= \{(1, 2, 5)\} \\
Or^{\mathcal{M}} &= \{(2, 3, 4)\} \\
Not^{\mathcal{M}} &= \{(5, 3)\}
\end{aligned}
$$

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

Example 4.7. 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:

And		
1	2	5

Or		
2	3	4

Atom
3
4

Not	
5	3

The parse tree T 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.

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 nodes in the parse tree correspond to which vocabulary symbols. A predicate tag, of the form `<predicate>`, is used to identify which nodes in the parse tree correspond to the relation described by the tag. The name of the relation is given by the name attribute, and the grammar rules which correspond to the relation are given by a grammar tag.

Example 4.8. The following predicate tag is used to describe the relation for the connective $\wedge$ in propositional logic.

```
<predicate name="And">
    <grammar>And</grammar>
    <arity>(Formula,Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
```

The arity tag specifies the arity of the relation and the typing of its tuples' arguments. The vocabulary symbols used within the arity tag must be described by a type tag in the vocabulary map. The vocabulary symbols described in the type tags assign typing to the nodes of the parse tree. The typing assigned to the tuples by the arity tag will correspond to the typing of a node corresponding to the relation and its child nodes. The format tag specifies how the contents of the relation's tuples are determined. In the example for $\wedge$, keyword `number_children` indicates that the first argument is the identifier of the node representing the element and the remaining arguments are the identifiers of its children.

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

Example 4.9. The predicate tag

```
<predicate name="Atom">
    <grammar>Atom</grammar>
    <arity>(Formula)</arity>
    <format>number</format>
    <terminal/>
</predicate>
```

represents the atoms of a formula.

The predicate identifying the atoms of the formula has only one argument 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`.

Example 4.10. The following is the complete vocabulary map file for propositional logic

```
<propositional structure="Instance" vocabulary="Propositional">
    <type name="Formula">
        <grammar>And,Or,Not,Atom</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <predicate name="And">
        <grammar>And</grammar>
        <arity>(Formula,Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Or">
        <grammar>Or</grammar>
        <arity>(Formula,Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Not">
        <grammar>Not</grammar>
        <arity>(Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Atom">
        <grammar>Atom</grammar>
        <arity>(Formula)</arity>
        <format>number</format>
        <terminal/>
    </predicate>
</propositional>
```

To view the complete syntax of vocabulary map files see Appendix A.

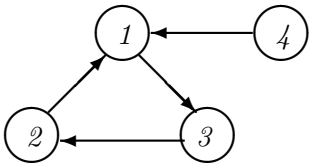
4.3 Graph Problems

Our method can easily be used to translate graphs presented in many formats. In graph applications, DIMACS formats are often used for storing graphs in files. In this section, we describe grammars and vocabulary maps for three of the DIMACS graph formats. The grammars and vocabulary maps, along with a corresponding problem specifications, can be

used the solve a variety of graph problems. We will also introduce the concepts of *inttypes*, invisible relations and vocabulary symbols which do not correspond to a grammar rule.

4.3.1 Graphs

Consider the following graph G as shown in Chapter 2.



Here is an example of graph G, as represented in the DIMACS graph format [10]. The first line gives the numbers of edges and nodes. Each line beginning with an 'e' gives an edge.

```
p edge 4 4
e 3 2
e 2 1
e 1 3
e 4 1
```

The DIMACS graph format is described by the following grammar.

```
Problem -> "p" "edge" Num Num Graph
Graph -> Edg | Node | Edg Graph | Node Graph
Edg -> "e" Vtx Vtx
Vtx -> _integer[1:VtxMAX]
Num -> _integer[intMIN:intMAX]
```

The vocabulary corresponding to graph G is

$$\tau_{Graph} = [\text{Edge}]$$

and the τ_{Graph} structure $\mathcal{G}$ corresponding to our graph G has:

$$V = \{1, 2, 3, 4, 5\}$$
$$Edge^{\mathcal{G}} = \{(3, 2), (2, 1), (1, 3), (4, 1)\}$$

In the τ_{Graph} structure $\mathcal{G}$ the set of vertices is our universe V, and therefore there is no vocabulary symbol which corresponds to the grammar rule `Vtx`. However, we still need to assign identifiers to the vertices, so we cannot ignore nodes which correspond to the grammar rule `Vtx`. Instead we will create an invisible relation, indicated by the invisible tag appearing in the predicate tag describing the relation `Vertex`.

The following is a corresponding vocabulary map for the grammar.

```
<graph structure="Instance" vocabulary="Graph">
    <type name="Vtx">
        <grammar>Vtx</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <predicate name="Vertex">
        <grammar>Vtx</grammar>
        <arity>(Vtx)</arity>
        <format>number</format>
        <terminal/>
        <invisible/>
    </predicate>
    <predicate name="Edge">
        <grammar>Edg</grammar>
        <arity>(Vtx,Vtx)</arity>
        <format>children</format>
    </predicate>
</graph>
```

Here we introduce a new keyword `children`, used in the format tag of predicate `Edge`. The keyword indicates that for nodes of type `Edg` the arguments of its tuples are the identifiers of the node's children. We give an example of a graph problem which uses the grammar and vocabulary map for DIMCAS graph format in Section 4.3.1.

Example Graph Problem: Colouring

One problem which uses the grammar and vocabulary map from Section 4.3.1 is the Graph Colouring problem as described in Chapter 2. If the problem specification requires that the number of colours is given in the solver instance, this may be added by the vocabulary map, as follows.

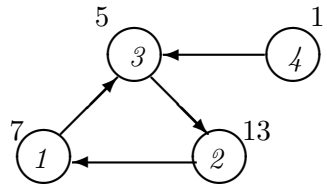
```
 <type name="Clr">
    <grammar></grammar>
    <range>
        <lower>1</lower>
        <upper>3</upper>
    </range>
```

```
    <inttype/>
</type>
```

The number of colours is provided by the range of type `Clr`, which has an empty grammar tag. This is a good way of including additional information required by the solver which is not provided by the instance string. For a complete vocabulary map for the Graphs Colouring problem see Appendix B.

4.3.2 Graphs with Weighted Nodes

Consider the graph G with weighted nodes.



Here is an example of graph G, as represented in the DIMACS graph format with weighed nodes [10]. Each line beginning with an 'n' gives a node number and its weight.

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

To create a grammar for these graphs, modify the graph grammar of Section 4.3.1 by replacing the `Graph` grammar rule and adding the following grammar rules for nodes with weights.

```
Graph -> Node | Node Graph | Edg | Edg Graph
Node -> "n" Vtx Num
```

The vocabulary corresponding to graph G is

$$\tau_{Graph} = [\text{Edge}, \text{Weight}]$$

and the τ_{Graph} structure $\mathcal{G}$ corresponding to our graph G has:

$$V = \{1, 2, 3, 4\}$$

$$Edge^{\mathcal{G}} = \{(3,2),(2,1),(1,3),(4,1)\}$$
$$Weight^{\mathcal{G}} = \{(1,5),(2,7),(3,13),(4,1)\}$$

To create a corresponding vocabulary map add the following type tag and predicate tag to the vocabulary map of Section 4.3.1.

```
<type name="Num">
    <grammar>Num</grammar>
    <range>
        <lower>IntMIN</lower>
        <upper>IntMAX</upper>
    </range>
    <inttype/>
</type>
<predicate name="Weight">
    <grammar>Node</grammar>
    <arity>(Vtx,Num)</arity>
    <format>children</format>
</predicate>
```

Here we introduce the concept of an *inttype*, an *inttype*, indicated by the tag `<inttype/>`, specify nodes of the parse tree which represent a numerical value. The nodes of the parse tree which are *inttypes* are treated differently then other nodes. Instead of assigning identifiers incrementally *inttype* nodes are assigned the value it represents as an identifier.

Consider the nodes in the parse tree for graph G which represents the weight of the vertices, such as `n 1 5`. Instead of assigning the weights the identifiers *1-4*, which would be the case for a regular type, the nodes will be assigned the identifiers *5,7,13,1* respectively.

While in most cases it is best not to set an upper bound for identifiers, *inttypes* are an exception. An upper bound is require for *inttypes* because the greatest value assigned to the identifiers will be greater then the number of nodes of this type in almost all cases. For a complete grammar and vocabulary map for graphs with weighted nodes see Appendix B.

Example Graph Problem: Weighted Clique

The grammar and vocabulary map from Section 4.3.2, along with a corresponding problem specification, can be used the solve the Weighted Clique problem.

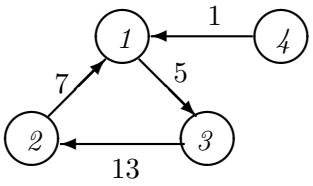
To find a clique of a weight greater than K in graph G we can use the following problem specification.

$$\forall x \forall y[(Clique(x) \wedge Clique(y)) \supset ((x = y) \vee Edge(x,y) \vee Edge(y,x))]$$

$$\wedge[Sum\{w : Weight(v, w)|Clique(v)\} \geq K]$$

4.3.3 Graphs with Weighted Edges

Consider the graph G with weighted edges.



Here is an example of graph G, as represented in the DIMACS graph format with weighted edges [9]. Each line beginning with an 'e' gives an edge (the first two numbers) and its weight (the third number).

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

To create a grammar for these graph replace the grammar rule describing edges in the grammar of Section 4.3.1 with the following grammar rule describing weighted edges.

```
Edg -> "e" Vtx Vtx Num
```

The vocabulary corresponding to graph G is

$$\tau_{Graph} = [\text{Edge}]$$

and the τ_{Graph} structure $\mathcal{G}$ corresponding to our graph G has:

$$V = \{1, 2, 3, 4\}$$
$$Edge^{\mathcal{G}} = \{(3, 2, 13), (2, 1, 7), (1, 3, 5), (4, 1, 1)\}$$

To create a corresponding vocabulary map remove the `Weight` predicate tag from the vocabulary map of Section 4.3.2 and replace the `Edge` predicate tag with the following predicate tag.

```
<predicate name="Edge">
    <grammar>Edg</grammar>
    <arity>(Vtx,Vtx,Num)</arity>
    <format>children</format>
</predicate>
```

The grammar and vocabulary map, along with a corresponding problem specification, can be used the solve a variety of graph problems. For a complete grammar and vocabulary map for graphs with weighted edges see Appendix B.

4.4 Logics

Our method can be applied to formulas of a variety of logics. In this section we describe grammars and vocabulary maps for several simple logics, each demonstrating a different sort of extension of our grammar and vocabulary map for propositional logic. We also introduce the concepts of sub-trees as terminals and multiple relations corresponding to a single grammar rule.

4.4.1 Integer Difference Logic

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

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

We can produce a grammar for this logic by modifying our grammar for propositional logic, as given in Section 4.1, replacing the `Atom` grammar rule with the following rules for the more complex atoms.

```
Atom -> "(" LessThan ")" | "(" Equal ")" | "(" GreaterThan ")"
LessThan -> Variable "-" Variable "<" Constant
Equal -> Variable "-" Variable "=" Constant
GreaterThan -> Variable "-" Variable ‘‘>" Constant
Variable -> _character[lower]
Constant -> _integer[0:IntMAX]
```

The vocabulary for formulas of difference logic is

$$\tau_{IDL} = [\text{And}, \text{Or}, \text{Not}, \text{LessThan}, \text{Equal}, \text{GreaterThan}]$$

and the τ_{IDL} structure $\mathcal{M}$ corresponding to our formula I has:

$$M = F \cup V$$
$$F = \{1, 2, 3, 4, 5, 6\}$$
$$V = \{1, 2, 3\}$$
$$And^{\mathcal{M}} = \{(1, 2, 5)\}$$

$$Or^{\mathcal{M}} = \{(2,3,4)\}$$
$$Not^{\mathcal{M}} = \{(5,6)\}$$
$$LessThan^{\mathcal{M}} = \{(3,1,2,9)\}$$
$$Equal^{\mathcal{M}} = \{(4,1,2,9)\}$$
$$GreaterThan^{\mathcal{M}} = \{(6,2,3,6)\}$$

where atoms are separated into three different relations based on their operation (=,<,>). The tuples of these three relations have four arguments. The first argument is the identifier of the sub-formula, the second and third arguments are the identifiers of the atom's variables and the final argument is the value of the atom's constant. For example, the atom $(x-y<9)$ is in relation $LessThan$ and is represented by tuple $(3,1,2,9)$.

To create a vocabulary map remove the `Formula` type and `Atom` predicate from the vocabulary map for propositional logic and add the following tags.

```
<type name="Formula">
    <grammar>And,Or,Not,LessThan,Equal,GreaterThan</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Variable">
    <grammar>Variable</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Number">
    <grammar>Constant</grammar>
    <range>
        <lower>0</lower>
        <upper>intMAX</upper>
    </range>
    <inttype/>
</type>
<predicate name="LessThan">
    <grammar>LessThan</grammar>
    <arity>(Formula,Variable,Variable,Number)</arity>
    <format>number_children</format>
    <terminal/>
```

```
</predicate>
<predicate name="Equal">
    <grammar>Equal</grammar>
    <arity>(Formula,Variable,Variable,Number)</arity>
    <format>number_children</format>
    <terminal/>
</predicate>
<predicate name="GreaterThan">
    <grammar>GreaterThan</grammar>
    <arity>(Formula,Variable,Variable,Number)</arity>
    <format>number_children</format>
    <terminal/>
</predicate>
<predicate name="Variable">
    <grammar>Variable</grammar>
    <arity>(Variable)</arity>
    <format>number</format>
    <terminal/>
    <invisible/>
</predicate>
```

This grammar and vocabulary map, with a corresponding problem specification, can be used to solve for integer difference logic satisfiability. For a complete grammar and vocabulary map for integer difference logic see Appendix C.

4.4.2 Propostional Formulas in Conjunctive Normal Form

Many propositional logic formulas are written in Conjunctive Normal Form (CNF). A formula in CNF consists of a conjunction of clauses where each clause is a disjunction of literals and a literal is an atom or a negated atom.

Example 4.11. Let C be the formula

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

The atoms of C are p, q and r, the literals are p, q, r, $\neg p$, $\neg q$ and $\neg r$ and the clauses are $(p \vee q \vee \neg r)$, $(\neg p \vee q \vee r)$ and $(p \vee \neg q \vee \neg r)$.

The following is a general grammar file for propositional formulas in CNF:

```
Conj -> "(" Disj ")" | "(" Disj ")" "&" Conj
Disj ->  Atom | Neg |  Atom "|" Disj | Neg "|" Disj
```

```
Neg -> "-" Atom
Atom -> _character[lower]
```

The vocabulary for propositional formulas in CNF is

$$\tau_{CNF} = [\text{Clause}, \text{Literal}, \text{Not}, \text{Atom}]$$

and the τ_{CNF} structure $\mathcal{M}$ corresponding to our formula C has:

$$M = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$
$$Clause^{\mathcal{M}} = \{(1), (6), (8)\}$$
$$Literal^{\mathcal{M}} = \{(1, 2), (1, 3), (1, 4), (6, 7), (6, 3), (6, 5), (8, 2), (8, 9), (8, 10)\}$$
$$Not^{\mathcal{M}} = \{(4, 5), (6, 2), (9, 3), (9, 5)\}$$
$$Atom^{\mathcal{M}} = \{(2), (3), (5)\}$$

It is possible to use the propositional logic grammar of Section 4.1 to parse propositional formulas in CNF. However, by taking advantage of the restricted format of CNF we can produce instance structures with much smaller universes then those produced by our method for general formulas.

Example 4.12. The τ_{PL} structure corresponding to our formula C would have universe $M = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$, where every sub-formula of C is assigned an identifier. The τ_{CNF} structure $\mathcal{M}$ corresponding to our formula C only has a universe with ten identifiers.

The corresponding vocabulary map is the following:

```
<cnf structure="Instance" vocabulary="CNF">
    <type name="Formula">
        <grammar>Disj,Neg,Atom</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <predicate name="Clause">
        <grammar>Disj</grammar>
        <arity>(Formula)</arity>
        <format>number</format>
    </predicate>
    <predicate name="Literal">
        <grammar>Disj</grammar>
```

```
            <arity>(Formula,Formula)</arity>
            <format>number_child</format>
        </predicate>
        <predicate name="Not">
            <grammar>Neg</grammar>
            <arity>(Formula,Formula)</arity>
            <format>number_children</format>
        </predicate>
        <predicate name="Atom">
            <grammar>Atom</grammar>
            <arity>(Formula)</arity>
            <format>number</format>
            <terminal/>
        </predicate>
    </cnf>
```

Here we introduce the concept of multiple relations corresponding to a single grammar rule. Both the relation `Clause` and the relation `Literal` correspond to the grammar rule `Disj`. The relation `Clause` indicates which sub-formulas are clauses. The relation `Literal` describes the relationship between clauses and literals. It is possible to combine the two relations into a new single relation, however, this would require a more complex problem specification.

Example 4.13. Consider the constraint stating that all clauses must be true for the formula to be satisfied. When using two relations the constraint is written as follows:

$$\forall c[Clause(c) \supset True(c)]$$

The relation `Literal` could be used be used to indicate which sub-formulas are clauses and to describe the relationships between clauses and literals. When using only one relation the constraint would be the following.

$$\forall c \forall l[Literal(c, l) \supset True(c)]$$

When the first version of the constraint is ground there will be one version of the constraint for each clause. However, when the second version of the constraint is ground there will be one version of the constraint for each literal and thus multiple versions of the constraint for each clause. These additional constraints increase the complexity of the specification and can increase the runtime of the problem solver.

In order to keep problem specifications as simple as possible the two separate relations are created instead. We also introduce a new keyword `number_child` within the format tag

of predicate `Literal`. This keyword indicates that for nodes of type `Disj` a tuple is created for each of the node's children where the arguments of the tuple are the identifiers of the node and its child.

The following is a specification for propositional logic satisfiability for formulas in CNF.

$$\begin{aligned} &\forall c[Clause(c) \supset True(c)] \\ \wedge &\forall c[Clause(c) \supset (True(c) \leftrightarrow (\exists l Literal(c,l) \wedge True(l)))] \\ \wedge &\forall n \forall a[Not(n,a) \supset (True(n) \leftrightarrow \neg True(a))] \end{aligned}$$

The standard input format for SAT solvers is CNF formulas in DIMACS format [10]. In this format, the first line gives the number of atoms and the number of clauses and each line following is a clause. A clause is a list of literals ending with `0`, a literal is an integer where positive integers are atoms and negative integers are negated atoms.

Example 4.14. Our formula C in DIMACS format is

```
p cnf 3 3
1 2 0
-1 3 0
-2 -3 0
```

The following grammar describes the DIMACS format for CNF formulas. This new grammar can also be used with the vocabulary map and specification for CNF formulas given above.

```
Problem -> "p" "cnf" Num Num Conj
Conj -> Disj "0" | Disj "0" Conj
Disj ->  Atom | Atom Disj | Neg | Neg Disj
Atom -> _integer[1:intMAX]
Neg -> "-" Atom
Num -> _integer[0:IntMAX]
```

4.4.3 Ground Function-Free First Order Logic

Ground function-free first order logic is, syntactically, propositional logic where atoms are in the form of a predicate symbol applied to a tuple of constant symbols.

Example 4.15. Let F be the formula

$$((Q(b) \vee Q(c)) \wedge (\neg P(a,b)))$$

where P and Q are predicates and a, b and c are constants.

Satisfiability of these formulas can be decided by treating each distinct ground atom as a distinct propositional atom. As a consequence, if our goal is satisfiability checking, we can modify the grammar for propositional logic given in Section 4.1 to deal with the new form of atoms, and then use the same vocabulary map and specification we used for propositional logic.

To create a grammar for ground function-free first order formulas replace the `Atom` grammar rule in the grammar for Proposition logic with the following grammar rules.

```
Atom -> Pred "(" Cons ")"
Pred -> _character[upper]
Cons -> Con | Con "," Cons
Con -> _character[lower]
```

The vocabulary corresponding to formula F is

$$\tau_{GFO} = [\text{And}, \text{Or}, \text{Not}, \text{Atom}]$$

and the τ_{GFO} structure $\mathcal{M}$ corresponding to our formula F has:

$$\begin{aligned} M &= \{1, 2, 3, 4, 5, 6\} \\ And^{\mathcal{M}} &= \{(1, 2, 5)\} \\ Or^{\mathcal{M}} &= \{(2, 3, 4)\} \\ Not^{\mathcal{M}} &= \{(5, 6)\} \\ Atom^{\mathcal{M}} &= \{(3), (4), (6)\} \end{aligned}$$

Here we introduce the concept of sub-trees as terminals. In the case where the root of a sub-tree is a semantically meaningful element of the parse tree and all the nodes below the root are not semantically meaningful the whole sub-tree can be treated as a terminal node. We treat each sub-tree rooted at such a node as if the whole tree is a terminal symbol, with each distinct tree of this sort being treated as if it was a distinct terminal symbol.

Example 4.16. The sub-formula $P(a, b)$ of formula F is represented by a node of type `Atom` in the parse tree. The `Atom` node has child nodes of types `Pred`, which presents the predicate P, and `Cons`, which will have two child nodes of types `Con` representing the constants a and b.

When used with the vocabulary map and specification for proposition logic this grammar can be used to determine satisfiability for ground first order formulas. For a complete grammar and vocabulary map for ground first order formulas see Appendix C.

4.4.4 Function-Free First Order Logic

Formulas of function-free first order logic have atoms consisting of predicate symbols applied to a tuple of variable symbols, and the quantifiers $\forall$ and $\exists$.

Example 4.17. Let F be the formula

$$(\forall y((\forall z(Q(y) \vee Q(z))) \wedge (\exists x(\neg P(x, y)))))$$

where P and Q are predicates and x, y and z are variable symbols.

To create a grammar for first order formulas, from the grammar for ground first order formulas, remove the grammar rules `Formula`, `Atom`, `Cons` and `Con` and add the following grammar rules.

```
Formula -> Atom | "(" Not ")" | "(" Or ")" |
            "(" And ")" | "(" Forall ")" | "(" Exists ")"
Forall -> "!" Var Formula
Exists -> "?" Var Formula
Atom -> Pred "(" Vars ")"
Pred -> _character[upper]
Vars -> Var | Var "," Vars
Var -> _character[lower]
```

The vocabulary for function-free first order formulas is

$$\tau_{FO} = [\text{Forall}, \text{Exists}, \text{And}, \text{Or}, \text{Not}, \text{Atom}, \text{Variable}]$$

and the τ_{FO} structure $\mathcal{M}$ corresponding to our formula F has:

$$\begin{aligned}
&M = F \cup V \cup P \cup L \\
&F = \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
&V = \{1, 2, 3\} \\
&P = \{1, 2\} \\
&L = \{1, 2, 3\} \\
&Forall^{\mathcal{M}} = \{(1, 1, 2), (2, 3, 4)\} \\
&Exists^{\mathcal{M}} = \{(3, 7, 8)\} \\
&And^{\mathcal{M}} = \{(2, 3, 7)\} \\
&Or^{\mathcal{M}} = \{(4, 5, 6)\} \\
&Not^{\mathcal{M}} = \{(8, 9)\} \\
&Atom^{\mathcal{M}} = \{(5, 1, 1), (6, 1, 2), (9, 2, 3)\}
\end{aligned}$$

$$Variable^{\mathcal{M}} = \{(1,1),(2,2),(3,3),(3,1)\}$$

where atoms are represented by a ternary relation. The first argument is the identifier of the sub-formula, the second argument is the identifier of the atom's predicate and the third argument is the identifier of the atom's list of variables. The relation *Variable* describes the relationship between the atoms' list of variables and the variables themselves. The relation has two arguments; the first argument is the identifier of an atom's list of variables and the second argument is the identifier of the variable which is a member of the list. For example, the atom $P(x, y)$ is represented by the tuples $(9, 2, 3)$ in *Atom* and $(3, 3), (3, 1)$ in *Variable*.

To create a vocabulary map for function-free first order formulas remove the `Formula` type tag and the `Atom` predicate tag from the vocabulary map for propositional logic of Section 4.1 and add the following tags.

```
<type name="Formula">
    <grammar>Forall,Exists,And,Or,Not,Atom</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Var">
    <grammar>Var</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Var_List">
    <grammar>Vars</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Pred">
    <grammar>Pred</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<predicate name="Forall">
    <grammar>Forall</grammar>
    <arity>(Formula,Var,Formula)</arity>
```

```
    <format>number_children</format>
</predicate>
<predicate name="Exists">
    <grammar>Exists</grammar>
    <arity>(Formula,Var,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Atom">
    <grammar>Atom</grammar>
    <arity>(Formula,Pred,Var_List)</arity>
    <format>number_children</format>
    <terminal/>
</predicate>
<predicate name="Variable">
    <grammar>Vars</grammar>
    <arity>(Var_List,Var)</arity>
    <format>number_child</format>
</predicate>
<predicate name="Varble">
    <grammar>Var</grammar>
    <arity>(Var)</arity>
    <format>number</format>
    <terminal/>
    <invisible/>
</predicate>
<predicate name="Predicate">
    <grammar>Pred</grammar>
    <arity>(Pred)</arity>
    <format>number</format>
    <terminal/>
    <invisible/>
</predicate>
```

Here we have two invisible relations `Varble` and `Predicate` which are used to assign identifiers to nodes of type `Var` and `Pred` respectively. The relation `Variable` is used to describe the relationship between nodes of type `Vars` and nodes of type `Var`. For a complete grammar and vocabulary map for function-free first order formulas see Appendix C.

4.4.5 Modal Logic

Propositional modal logic extends the syntax of propositional logic the modal operators $\Box$ and $\Diamond$. Syntactically, these are similar to unary connectives. For example

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

To create a grammar for modal logic, modify the grammar for propositional logic of Section 4.1 by removing the grammar rules `Formula` and adding the following grammar rules.

```
Formula -> Atom | "(" Not ")" | "(" Or ")"
          | "(" And ")" | "(" Box ")" | "(" Dia ")"
Box -> "box" Formula
Dia -> "dia" Formula
```

The vocabulary corresponding to formula F is

$$\tau_{ML} = [\text{Box}, \text{Dia}, \text{And}, \text{Or}, \text{Not}, \text{Atom}]$$

and the τ_{ML} structure $\mathcal{M}$ corresponding to our formula F has:

$$\begin{aligned}
M &= \{1, 2, 3, 4, 5, 6, 7\} \\
Box^{\mathcal{M}} &= \{(5, 6)\} \\
Dia^{\mathcal{M}} &= \{(1, 2)\} \\
And^{\mathcal{M}} &= \{(6, 7, 4)\} \\
Or^{\mathcal{M}} &= \{(2, 3, 5)\} \\
Not^{\mathcal{M}} &= \{(3, 4)\} \\
Atom^{\mathcal{M}} &= \{(4), (7)\}
\end{aligned}$$

To create a corresponding vocabulary map, remove the `Formula` type tag and add the following tags.

```
<type name="Formula">
    <grammar>And,Or,Not,Box,Dia,Atom</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<predicate name="Box">
    <grammar>Box</grammar>
```

```
    <arity>(Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Dia">
    <grammar>Dia</grammar>
    <arity>(Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
```

For a complete grammar and vocabulary map for modal logic see Appendix C.

4.5 Declarative Solver Instances

Often there will be instances for the same problem represented in different solver instance formats. The instances can all be mapped to the same abstract structure by writing grammars for each solver instance format and a single vocabulary map for the instances.

Consider the Graph Colouring problem and our graph G from Chapter 2. For each declarative solver introduced in Chapter 2 we will show the solver instance of our graph G and a grammar for the solver instance (except Alloy which uses the DIMACS graph format). Every grammar shown below can be used with the vocabulary map for the Graph Colouring problem from Section 4.3.

4.5.1 Zimpl

The representation of graph G in Zimpl requires two files but only the file describing the edges of the graph is needed for mapping.

```
# Set E
1 3
2 1
3 2
4 1
```

The following is the grammar for graphs represented in Zimpl.

```
File -> "# Set E" Graph
Graph -> Edg | Edg Graph
Edg -> Vtx Vtx
Vtx -> _integer[1:IntMAX]
```

4.5.2 MiniZinc

The following is the representation of graph G for MiniZinc

```
sizeV = 4;
sizeE = 4;
sizeC = 3;
E = [ | 1, 3 | 2, 1 | 3, 2 | 4, 1 | ];
```

and the grammar for graphs represented in this format.

```
File -> V E C Graph
V -> "sizeV =" Num ";"
E -> "sizeE =" Num ";"
C -> "sizeC =" Num ";"
Graph -> "E = [" Edges "];"
Edges -> "|" Edg "|" | "|" Edg Edges
Edg -> Vtx "," Vtx
Vtx -> _integer[1:IntMAX]
Num -> _integer[0:IntMAX]
```

4.5.3 IDP System

The following is the IDP System representation of graph G.

```
structure Graph:Colouring {
 V = {1..4}
 C = {1..3}
 E = {(1,3);(2,1);(3,2);(4,1)}
}
```

The grammar for graphs represented for the IDP System is the following.

```
File -> "structure Graph:Colouring {" Graph "}"
Graph -> V C E
V -> "V = {" Num ".." Num "}"
C -> "C = {" Num ".." Num "}"
E -> "E = {" Edges "}"
Edges -> Edg | Edg ";" Edges
Edg -> "(" Vtx "," Vtx ")"
Vtx -> _integer[1:IntMAX]
Num -> _integer[0:IntMAX]
```

4.5.4 Enfragmo

The following is the representation of graph G for Enfragmo

```
TYPE V [1..4]
TYPE C [1..3]
PREDICATE E
 (1,3)
 (2,1)
 (3,2)
 (4,1)
```

and the grammar for graphs represented in Enfragmo.

```
Graph -> V C E
V -> "TYPE V [" Num ".." Num "]"
C -> "TYPE C [" Num ".." Num "]"
E -> "PREDICATE E" Edges
Edges -> Edg | Edg Edges
Edg -> "(" Vtx "," Vtx ")"
Vtx -> _integer[1:IntMAX]
Num -> _integer[0:IntMAX]
```

4.5.5 clingo

The graph G is represented for clingo as follows

```
node(1..4).
edge(1,3). edge(2,1). edge(3,2). edge(4,1).
```

and the grammar for graphs represented in this format is the following.

```
Graph -> Node | Edg | Node Graph | Edg Graph
Node -> "node(" Num ".." Num ")."
Edg -> "edge(" Vtx "," Vtx ")."
Vtx -> _integer[1:IntMAX]
Num -> _integer[0:IntMAX]
```

Chapter 5

Using Instance Structures

In this chapter we describe how to store and use instance structures. In Section 5.1 we describe the representation of instance structures used by the INSTLATOR system. Section 5.2 describes the method for mapping instance structures to solver instances. In Section 5.3 we describe how to write problem specifications for use with solver instances.

5.1 Storing Instance Structures

For storing an instance structure in a file, we use an XML representation. The XML representation of an instance structure contains all the information necessary to generate a solver instance and write a problem specification for a declarative solver using the instance vocabulary.

The instance vocabulary and the instance structure are both recorded in a single XML representation. The format for the XML representation has two main tags, one to record the vocabulary and one to record the structure itself, with the overall representation being of the form:

```
<instance>
    <vocabulary .... </vocabulary>
    <structure>... </structure>
</instance>
```

The instance vocabulary consists of a domain and relations. The domain is stored in the domain tag which contains a type tag with a name attribute for each type in the instance structure. The relations are stored in the relations tag which contains a relation tag for each relation in the instance structure. Each relation tag has a name attribute and an arity tag which gives the arity and typing of the arguments of the relation.

Example 5.1. Consider our vocabulary for propositional logic from Section 4.1.

$$\tau_{PL} = [\text{Atom}, \text{Not}, \text{And}, \text{Or}]$$

The XML representation of τ_{PL} is the following.

```
<vocabulary name="Propositional">
    <domain>
        <type name="Formula"/>
    </domain>
    <relations>
        <relation name="And">
            <arity>(Formula,Formula,Formula)</arity>
        </relation>
        <relation name="Or">
            <arity>(Formula,Formula,Formula)</arity>
        </relation>
        <relation name="Not">
            <arity>(Formula,Formula)</arity>
        </relation>
        <relation name="Atom">
            <arity>(Formula)</arity>
        </relation>
    </relations>
</vocabulary>
```

The instance structure consists of types and predicates which correspond to the vocabulary domain and relations. The types are stored in the types tag which contains a type tag for each type in the structure. Each type tag has a name attribute and a lower and upper tag which give the lower and upper bound for the type's identifiers. The predicates are stored in the predicates tag which contains a predicate tag for each predicate in the structure. Each predicate tag has a name attribute, a format tag which specifies how the content of tuples is determined and a tuples tag which contains a tuple tag for each of the predicate's tuples.

Example 5.2. Consider the τ_{PL} structure $\mathcal{M}$ for our formula F from Section 4.2.

$$M = \{1, 2, 3, 4, 5\}$$
$$Atom^{\mathcal{M}} = \{(3), (4)\}$$
$$And^{\mathcal{M}} = \{(1, 2, 5)\}$$
$$Or^{\mathcal{M}} = \{(2, 3, 4)\}$$
$$Not^{\mathcal{M}} = \{(5, 3)\}$$

The XML representation of τ_{PL} structure $\mathcal{M}$ is the following.

```
<structure name="Instance">
    <types>
        <type name="Formula">
            <lower>1</lower>
            <upper>5</upper>
        </type>
    </types>
    <predicates>
        <predicate name="And">
            <format>number_children</format>
            <tuples>
                <tuple>(1, 2, 5)</tuple>
            </tuples>
        </predicate>
        <predicate name="Or">
            <format>number_children</format>
            <tuples>
                <tuple>(2, 3, 4)</tuple>
            </tuples>
        </predicate>
        <predicate name="Not">
            <format>number_children</format>
            <tuples>
                <tuple>(5, 3)</tuple>
            </tuples>
        </predicate>
        <predicate name="Atom">
            <format>number</format>
            <tuples>
                <tuple>(3,)</tuple>
                <tuple>(4,)</tuple>
            </tuples>
        </predicate>
    </predicates>
</structure>
```

To view the complete syntax of instance structure files see Appendix A.

5.2 Mapping Structures to Solver Instances

The second stage of our method maps instance structures to solver instances using a solver format description. In the INSTLATOR system, this is done by mapping an instance structure file to an instance file for a specific solver as determined by a solver format file. We will illustrate this using our running example, formula F, and the IDP System introduced in Chapter 2.

Example 5.3. Assume we have an IDP System specification describing the semantics of propositional logic, using the vocabulary τ_{PL}. 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 instance structure, as described in Section 4.2.

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

Example 5.4. Here is a solver format file for the IDP System.

```
<idp>
   <structure>
      structure $structure$:$vocabulary$ {\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 needed here), which consists of the name followed by an appropriately formatted list of elements, in this case those in the range from `$lower$` to `$upper$`. The relation tag specifies the format of descriptions of relations. In this case, that is the name, followed by an appropriately formatted list of tuples. To view the complete syntax of solver format files see Appendix A.

5.3 Using Problem Specifications with Solver Instances

Once a solver instance has been created it is used along with a problem specification, written in the declarative language of the solver, to find a solution to the required problem.

Example 5.5. Our grammar for propositional logic is given in Section 4.1. For a vocabulary map that maps formulas to structures as described in Section 4.2, the following first order formula constitutes a specification of standard propositional logic satisfiability.

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

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.

Example 5.6. The following specification defines a standard V-valued propositional logic, for any positive integer V.

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

$$\begin{aligned}
&\wedge Value(f', v') \wedge (v = (V - v')))] \\
&\wedge Value(F, V)
\end{aligned}$$

Many other semantics for multi-valued logics are also possible. The following is a specification for Kleene 3-valued logic.

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

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

A problem specification can also be modified to solve new problems, as we did with grammars and vocabulary maps in Chapter 4.

Example 5.7. Consider our example of Integer Difference Logic from Section 4.4. By modifying our problem specification for propositional logic, in a similar way to the modification of our propositional grammar, we can write a problem specification for integer difference logic satisfiability.

$$\begin{aligned}
&\forall v \exists n[Value(v, n)] \\
&\wedge \forall v \forall n[Value(v, n) \supset \neg \exists n' < n(Value(v, n'))] \\
\wedge \forall f, f_1, f_2[&And(f, f_1, f_2) \supset
\end{aligned}$$

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

5.3.1 Solvers that Ground to SAT

Some declarative solvers work by grounding to SAT and running a SAT solver. These solvers transform a specification formula and instance structure into a ground formula that defines the solutions to the instance and then transform this into a formula of propositional logic in CNF. This CNF formula (which is normally in the DIMACS form described in Section 4.3), is sent to a SAT solver. Satisfying assignments for the formula correspond to expansions of the instance structure that constitute solutions. The transformation to CNF is normally carried out by a linear-time method that introduces new atoms.

It is interesting to consider what happens in the case that we apply our method to make such a system into a model finder for propositional logic, using the grammar and specification presented in Sections 4.1 and 5.2.

The grounding algorithms used by these systems are often complex, and the exact CNF formula generated by a particular system can only be determined by experiment (or study of the implementation). Instead, we consider what happens if we use the most direct form of grounding. In this method, we simply rewrite each sub-formula of the form $\forall x \psi$ as $\wedge_{a \in M} \psi[x \rightarrow a]$, and each sub-formula of the form $\exists x \psi$ as $\vee_{a \in M} \psi[x \rightarrow a]$, to obtain a ground formula, and then transform to CNF.

We may observe the following. Let ϕ be a formula of propositional logic, and suppose we generate an instance structure using our grammar and vocabulary map of Chapter 4, then ground the specification formula of Section 5.3 over that structure, and finally apply

the transformation to CNF. We obtain the same CNF formula that we get if we simply transform ϕ to CNF.

Example 5.8. Consider the τ_{PL} structure $\mathcal{M}$ for our formula F from Section 4.2 represented as a first-order formula.

$$And(1,2,5) \wedge Or(2,3,4) \wedge Not(5,3)$$

When we ground the specification formula of Section 5.3 over structure $\mathcal{M}$ we get

$$\begin{aligned} & And(1,2,5) \wedge Or(2,3,4) \wedge Not(5,3) \\ \wedge & And(1,2,5) \supset (True(1) \leftrightarrow (True(2) \wedge True(5))) \\ \wedge & Or(2,3,4) \supset (True(2) \leftrightarrow (True(3) \vee True(4))) \\ \wedge & Not(5,3) \supset (True(5) \leftrightarrow \neg True(3)) \\ & \wedge True(1) \end{aligned}$$

which is also the ground first-order representation of the parse tree T from Section 4.2.

When we apply the transformation to CNF we get a formula equivalent to the following

$$\begin{aligned} & (True(1) \leftrightarrow (True(2) \wedge True(5))) \\ \wedge & (True(2) \leftrightarrow (True(3) \vee True(4))) \\ \wedge & (True(5) \leftrightarrow \neg True(3)) \\ & \wedge True(1) \end{aligned}$$

which describes the relationships between the truth assignments of all sub-formulas of F. This formula is also equivalent to the formula obtained by transforming formula F to CNF where $p \equiv True(3)$, $q \equiv True(4)$ and $True(1)$,$True(2)$,$True(5)$ are equivalent to the new atoms introduced by the transformation.

Chapter 6

Conclusion

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 executable code. We have described here a method and a prototype system 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 method we introduced 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. Our method could also be used to create a solver-independent database of benchmark problem instances by storing XML representations of instance structures.

In the future, we will need to develop a theory to determine the range of applicability of our method. We also need to determine some measure of the generality of our languages for describing vocabulary maps and solver format descriptions. Currently, the method for mapping structures to solver instances does not have the generality that might be desired. The language for describing solver formats should be expanded to allow a wider range of solver instances to be described, including a wider range of domain and relation formats. We will also investigate whether compilation technology, such as attribute grammars, and source-to-source translation [18] might be fruitfully used for our task. The language-theoretic approach described in [4] may also be useful for our task.

The Instlator system will also require further work. The current parser, NLTK [13], imposes inconvenient restrictions and causes major performance problems, presumably because it is intended for natural language processing and was not designed to parse the sort of inputs we face, such as large CNF formulas or graphs. The system is intended for users with little programming experience, so future versions should include tools which assist users in creating grammars, vocabulary maps and solver format descriptions. Ideally, the

system should also have a graphical user interface for each tool to further improve usability for non-programmers.

Nonetheless, our experience in designing and building the INSTLATOR system 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.

Bibliography

[1] Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David G. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In Nikolaj Bjørner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 15–22. Springer, 2012.

[2] Continuum Analytics. *Anaconda Software Distribution*. Computer software. Vers. 2-2.4.0. Web. `https://continuum.io`, November 2015.

[3] Bart Bogaerts, Joachim Jansen, Broes De Cat, Gerda Janssens, Maurice Bruynooghe, and Marc Denecker. Meta-level representations in the IDP knowledge base system: Towards bootstrapping inference engine development, 2014. 2014 Workshop on Logic and Search (LaSh '14).

[4] Bruno Courcelle and Joost Engelfriet. *Graph structure and monadic second-order logic: a language-theoretic approach*, volume 138. Cambridge University Press, 2012.

[5] Herbert B. Enderton. *A mathematical introduction to logic*. Academic Press, 1972.

[6] M Gebser, R Kaminski, B Kaufmann, M Lindauer, M Ostrowski, J Romero, T Schaub, and S Thiele. Potassco user guide. *Institute for Informatics, University of Potsdam*, pages 330–331, 2015.

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

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

[9] David S Johnson, Catherine C McGeoch, et al. *Network flows and matching: first DIMACS implementation challenge*, volume 12. American Mathematical Soc., 1993.

[10] David S Johnson and Michael A Trick. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc., 1996.

[11] Thorsten Koch. *Rapid Mathematical Programming*. PhD thesis, Technische Universität Berlin, 2004. ZIB-Report 04-58.

[12] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[13] Edward Loper and Steven Bird. NLTK: the natural language toolkit. *CoRR*, cs.CL/0205028, 2002.

[14] D. G. Mitchell and E. Ternovska. Expressiveness and abstraction in ESSENCE. *Constraints*, 13(2):343–384, 2008.

[15] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 430–435. AAAI Press / The MIT Press, 2005.

[16] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, CP'07, pages 529–543, Berlin, Heidelberg, 2007. Springer-Verlag.

[17] F.C.N. Pereira and S.M. Shieber. *Prolog and Natural-Language Analysis*. Center for the Study of Language and Information Publication Lecture Notes. Cambridge University Press, 1987.

[18] David A Plaisted. Source-to-source translation and software engineering. 2013.

[19] James Rogers. A model-theoretic framework for theories of syntax. In Aravind K. Joshi and Martha Palmer, editors, *34th Annual Meeting of the Association for Computational Linguistics, 24-27 June 1996, University of California, Santa Cruz, California, USA, Proceedings.*, pages 10–16. Morgan Kaufmann Publishers / ACL, 1996.

[20] Lucas Swanson and Xiongnan (Newman) Wu. The Enfragmo system. `http://www.cs.sfu.ca/~mitchell/cmpt-827/2013-Fall/Project-Materials/Enfragmo/Enfragmo-man.pdf`, June 2010.

[21] Shahab Tasharrofi, Xiongnan (Newman) Wu, and Eugenia Ternovska. Solving modular model expansion: Case studies. In Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf, editors, *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*, volume 7773 of *Lecture Notes in Computer Science*, pages 215–236. Springer, 2011.

[22] Emina Torlak. Kodkod. `http://alloy.mit.edu/kodkod/`, November 2015.

[23] Johan Wittocx, Maarten Marien, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In *Proc., LaSh 2008*, 2008.

Appendix A

Syntax of Declarative Descriptions

A.1 Syntax of Grammar Descriptions

The following is the complete syntax of INSTLATOR grammar files.

```
<grammar_file> ::= <grammar_rule> | <grammar_rule> <grammar_file>
<grammar_rule> ::= <name> -> <description>
<name> ::= [a-zA-Z]+
<description> ::= <terminal> | <nonterminal>
                  | <terminal> `|' <description>
                  | <nonterminal> `|' <description>
<nonterminal> ::= <name> | "<string>" | <name> `|' <nonterminal>
                  | "<string>" `|' <nonterminal>
<terminal> ::= _character`['lower`]' | _character`['upper`]'
               | _number`['<number>`]' | _integer`['<integer>,<integer>`]'
               | "<string>"
<string> ::= [0-9a-zA-Z_]+
<number> ::= [0-9]+
<integer> ::= [-][0-9]+
```

A.2 Syntax of Vocabulary Map Descriptions

The following is the complete syntax of INSTLATOR vocabulary maps.

```
<vocabulary_map> ::= `<'<string> structure="<string>"
                     vocabulary="<string>"`>' <type_tags> <predicate_tags>
                     `<'/<string>`>'
<type_tags> ::= <type_tag> | <type_tag> <type_tags>
<type_tag> ::= `<'type name="<string>" `>' <type_description> `<'/type`>'
<predicate_tags> ::= `<'predicate`>' name="<string>"`>'
                      <predicate_description> `<'/predicate`>'
```

```
<type_description> ::= <grammar_tag> <range_tag>
                     | <grammar_tag> <range_tag> ‘<’inttype/‘>’
<predicate_description> ::= <grammar_tag> <arity_tag> <format_tag>
                          | <grammar_tag> <arity_tag> <format_tag>
                            <predicate_options>
<grammar_tag> ::= ‘<’grammar‘>’ <string> ‘<’/grammar‘>’
<range_tag> ::= ‘<’range‘>’ <lower_tag> ‘<’/range‘>’
              | ‘<’range‘>’ <lower_tag> <upper_tag>  ‘<’/range‘>’
<arity_tag> ::= ‘<’arity‘>’ (<string_list>) ‘<’/arity‘>’
<string_list> ::= <string> | <string>, <string_list>
<format_tag> ::= ‘<’format‘>’ <format> ‘<’/format‘>’
<format> ::= number | number_content | number_children | number_child
           | number_childrencontent | number_childcontent
           | content | content_children | content_child
           | content_childrencontent | content_childcontent
           | children | child | childrencontent | childcontent
<predicate_options> ::= ‘<’terminal/‘>’ | ‘<’invisible/‘>’
                      | ‘<’terminal/‘>’ ‘<’invisible/‘>’
<lower_tag> ::= ‘<’lower‘>’ <integer> ‘<’/lower‘>’
<upper_tag> ::= ‘<’upper‘>’ <integer> ‘<’/upper‘>’
<string> ::= [0-9a-zA-Z_]+
<integer> ::= [-][0-9]+
```

A.3 Syntax of Instance Structure Representations

The following is the complete syntax of instance structure files. All instance structure files produced by the INSTLATOR system use this syntax and the system will only accept instance structure files which conform to this syntax.

```
<instance> ::= ‘<’instance‘>’ <vocabulary> <structure> ‘<’/instance‘>’
<vocabulary> ::= ‘<’vocabulary name="<string>"‘>’ <domain_tag>
                  <relations_tag> ‘<’/vocabulary‘>’
<domain_tag> ::= ‘<’domain‘>’ <domain> ‘<’/domain‘>’
<relations_tag> ::= ‘<’relations‘>’ <relations> ‘<’/relations‘>’
<domain> ::=  <vocab_type> | <vocab_type> <domain>
<relations> ::=  <relation> | <relation> <relations>
<vocab_type> ::= ‘<’type name="<string>"/‘>’
<relations> ::= ‘<’relation name="<string>"‘>’ <arity> ‘<’/relation‘>’
<arity> ::= (<string_list>)
<structure> ::= ‘<’structure name="<string>"‘>’ <types_tag>
                <predicates_tag> ‘<’/structure‘>’
<types_tag> ::= ‘<’types‘>’ <types> ‘<’/‘types>’
<types> ::= <type> | <type> <types>
<type> ::= ‘<’type name="<string>"‘>’ <lower> <upper> ‘<’/type‘>’
<lower> ::= ‘<’lower‘>’ <integer> ‘<’/lower‘>’
<upper> ::= ‘<’upper‘>’ <integer> ‘<’/upper‘>’
```

```
<predicates_tag> ::= ‘<’predicates‘>’ <predicates> ‘<’/predicates‘>’
<predicates> ::= <predicate> | <predicate> <predicates>
<predicate> ::= ‘<’predicate name="<string>"‘>’ <format_tag>
                <tuples_tag> ‘<’/predicate‘>’
<format_tag> ::= ‘<’format‘>’ <format> ‘<’/format‘>’
<format> ::= number | number_content | number_children | number_child
           | number_childrencontent | number_childcontent
           | content | content_children | content_child
           | content_childrencontent | content_childcontent
           | children | child | childrencontent | childcontent
<tuples_tag> ::= ‘<’tuples‘>’ <tuples> ‘<’/tuples‘>’
<tuples> ::= <tuple> | <tuple> <tuples>
<tuple> ::= ‘<’tuple‘>’ (<integer_list>) ‘<’/tuple‘>’
<string_list> ::= <string> | <string>, <string_list>
<integer_list> ::= <integer> | <integer>, <integer_list>
<string> ::= [0-9a-zA-Z_]+
<integer> ::= [-][0-9]+
```

A.4 Syntax of Solver Format Descriptions

The following is the complete syntax of INSTLATOR solver format files.

```
<solver_format> ::= ‘<’<string>‘>’ <structure_tag> <domain_tag>
                     <relation_tag> ‘<’/<string>‘>’
<structure_tag> ::= ‘<’structure‘>’ <structure_description>
                    ‘<’/structure‘>’
<domain_tag> ::= ‘<’domain‘>’ <domain_description> ‘<’/domain‘>’
<relation_tag> ::= ‘<’relation separator="<string>"‘>’
                   <relation_description> ’<’/relation’>’
<structure_description> ::= <string> $structure$ <structure_description>
                          | <string> $vocabulary$ <structure_description>
                          | <string> $domains$ <structure_description>
                          | <string> $relations$ <structure_description>
                          | <string>
<domain_description> ::= <string> $name$ <domain_description>
                       | <string> $lower$ <domain_description>
                       | <string> $upper$ <domain_description>
                       | <string>
<relation_description> ::= <string> $name$ <relation_description>
                         | <string> $tuples$ <relation_description>
                         | <string>
<string> ::= [0-9a-zA-Z_ \n]+
```

Appendix B

Graph Problems

As shown in Chapter 4, our method can be used for a variety of graph problems. Here we provide complete implementations of the examples given in Chapters 4.

B.1 Graph Colouring

The following is a complete vocabulary map for the Graph Colouring problem as described in Chapter 4

```
<graph structure="Instance" vocabulary="Colouring">
    <type name="Vtx">
        <grammar>Vtx</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <type name="Clr">
        <grammar></grammar>
        <range>
            <lower>1</lower>
            <upper>3</upper>
        </range>
        <inttype/>
    </type>
    <predicate name="Vertex">
        <grammar>Vtx</grammar>
        <arity>(Vtx)</arity>
        <format>number</format>
        <terminal/>
    </predicate>
    <predicate name="Edge">
        <grammar>Edg</grammar>
```

```
        <arity>(Vtx,Vtx)</arity>
        <format>children</format>
    </predicate>
</graph>
```

B.2 Graphs with Weighted Nodes

As described in Chapter 4, the following is a complete grammar and vocabulary map for DIMACS graph format with weighted nodes.

```
Problem -> "p" "edge" Num Num Graph
Graph -> Node | Node Graph | Edg | Edg Graph
Node -> "n" Vtx Num
Edg -> "e" Vtx Vtx Num
Vtx -> _integer[1:IntMAX]
Num -> _integer[0:IntMAX]
```

The following is the complete corresponding vocabulary map.

```
<graph structure="Instance" vocabulary="Graph">
    <type name="Vtx">
        <grammar>Vtx</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <type name="Num">
        <grammar>Num</grammar>
        <range>
            <lower>0</lower>
            <upper>IntMAX</upper>
        </range>
        <inttype/>
    </type>
    <predicate name="Vertex">
        <grammar>Vtx</grammar>
        <arity>(Vtx)</arity>
        <format>number</format>
        <terminal/>
    </predicate>
    <predicate name="Edge">
        <grammar>Edg</grammar>
        <arity>(Vtx,Vtx)</arity>
        <format>children</format>
    </predicate>
    <predicate name="Weight">
```

```
        <grammar>Node</grammar>
        <arity>(Vtx,Num)</arity>
        <format>children</format>
    </predicate>
</graph>
```

B.3 Graphs with Weighted Edges

The complete grammar for DIMACS graph format with weighted edges as described in Chapter 4 is as follows.

```
Problem -> "p" "edge" Num Num Graph
Graph -> Edg | Edg Graph
Edg -> "e" Vtx Vtx Num
Vtx -> _integer[1:IntMAX]
Num -> _integer[0:IntMAX]
```

The complete corresponding vocabulary map is provided below.

```
<graph structure="Instance" vocabulary="Colouring">
    <type name="Vtx">
        <grammar>Vtx</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <type name="Num">
        <grammar>Num</grammar>
        <range>
            <lower>0</lower>
            <upper>IntMAX</upper>
        </range>
        <inttype/>
    </type>
    <predicate name="Vertex">
        <grammar>Vtx</grammar>
        <arity>(Vtx)</arity>
        <format>number</format>
        <terminal/>
    </predicate>
    <predicate name="Edge">
        <grammar>Edg</grammar>
        <arity>(Vtx,Vtx,Num)</arity>
        <format>children</format>
    </predicate>
</graph>
```

Appendix C

Logics

As shown in Chapter 4, our method can be used for a variety of logics. Here we provide complete implementations of the examples given in Chapters 4.

C.1 Integer Difference Logic

As described in Chapter 4, the grammar for integer difference logic is a modified version of our propositional logic grammar. The following is the complete grammar file.

```
Formula -> Atom | "(" And ")" | "(" Or ")" | "(" Not ")"
And -> Formula "&" Formula
Or -> Formula "|" Formula
Not -> "~" Formula
Atom -> "(" LessThan ")" | "(" Equal ")" | "(" GreaterThan ")"
LessThan -> Variable "-" Variable "<=" Constant
Equal -> Variable "-" Variable "=" Constant
GreaterThan -> Variable "-" Variable ">=" Constant
Variable -> _character[lower]
Constant -> _integer[0:IntMAX]
```

The complete vocabulary map for integer difference logic is provided below.

```
<idl structure="Instance" vocabulary="IDL">
    <type name="Formula">
        <grammar>And,Or,Not,LessThan,Equal,GreaterThan</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
    <type name="Variable">
        <grammar>Variable</grammar>
        <range>
```

```
        <lower>1</lower>
    </range>
</type>
<type name="Number">
    <grammar>Constant</grammar>
    <range>
        <lower>0</lower>
        <upper>IntMAX</upper>
    </range>
    <inttype/>
</type>
<predicate name="And">
    <grammar>And</grammar>
    <arity>(Formula,Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Or">
    <grammar>Or</grammar>
    <arity>(Formula,Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Not">
    <grammar>Not</grammar>
    <arity>(Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="LessThan">
    <grammar>LessThan</grammar>
    <arity>(Formula,Variable,Variable,Number)</arity>
    <format>number_children</format>
    <terminal/>
</predicate>
<predicate name="Equal">
    <grammar>Equal</grammar>
    <arity>(Formula,Variable,Variable,Number)</arity>
    <format>number_children</format>
    <terminal/>
</predicate>
<predicate name="GreaterThan">
    <grammar>GreaterThan</grammar>
    <arity>(Formula,Variable,Variable,Number)</arity>
    <format>number_children</format>
    <terminal/>
</predicate>
<predicate name="Variable">
    <grammar>Variable</grammar>
    <arity>(Variable)</arity>
```

```
        <format>number</format>
        <terminal/>
        <invisible/>
    </predicate>
</idl>
```

C.2 Ground Function-Free First Order Logic

The following is the complete grammar for ground first order formula as described in Chapter 4.

```
Formula -> Atom | "(" Not ")" | "(" Or ")" | "(" And ")"
Not -> "~" Formula
Or -> Formula "|" Formula
And -> Formula "&" Formula
Atom -> Pred "(" Cons ")"
Pred -> _character[upper]
Cons -> Con | Con "," Cons
Con -> _character[lower]
```

C.3 Function-Free First Order Logic

The following is the complete grammar for first order formula as described in Chapter 4.

```
Formula -> Atom | "(" Not ")" | "(" Or ")" |
            "(" And ")" | "(" Forall ")" | "(" Exists ")"
Forall -> "!" Var Formula
Exists -> "?" Var Formula
Not -> "~" Formula
Or -> Formula "|" Formula
And -> Formula "&" Formula
Atom -> Pred "(" Vars ")"
Pred -> _character[upper]
Vars -> Var | Var "," Vars
Var -> _character[lower]
```

The following is the complete corresponding vocabulary map.

```
<first_order structure="Instance" vocabulary="FO">
    <type name="Formula">
        <grammar>Forall,Exists,And,Or,Not,Atom</grammar>
        <range>
            <lower>1</lower>
        </range>
```

```
</type>
<type name="Var">
    <grammar>Var</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Var_List">
    <grammar>Vars</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Pred">
    <grammar>Pred</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<predicate name="Forall">
    <grammar>Forall</grammar>
    <arity>(Formula,Var,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Exists">
    <grammar>Exists</grammar>
    <arity>(Formula,Var,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="And">
    <grammar>And</grammar>
    <arity>(Formula,Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Or">
    <grammar>Or</grammar>
    <arity>(Formula,Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Not">
    <grammar>Not</grammar>
    <arity>(Formula,Formula)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Atom">
    <grammar>Atom</grammar>
    <arity>(Formula,Pred,Var_List)</arity>
```

```
        <format>number_children</format>
        <terminal/>
    </predicate>
    <predicate name="Variable">
        <grammar>Vars</grammar>
        <arity>(Var_List,Var)</arity>
        <format>number_child</format>
    </predicate>
    <predicate name="Varble">
        <grammar>Var</grammar>
        <arity>(Var)</arity>
        <format>number</format>
        <terminal/>
        <invisible/>
    </predicate>
    <predicate name="Predicate">
        <grammar>Pred</grammar>
        <arity>(Pred)</arity>
        <format>number</format>
        <terminal/>
        <invisible/>
    </predicate>
</first_order>
```

C.4 Modal Logic

As described in Chapter 4, the following is the complete grammar for modal logic.

```
Formula -> Atom | "(" Not ")" | "(" Or ")"
          | "(" And ")" | "(" Box ")" | "(" Dia ")"
Not -> "~" Formula
Or -> Formula "v" Formula
And -> Formula "&" Formula
Box -> "box" Formula
Dia -> "dia" Formula
Atom -> _character[lower]
```

The following is the complete vocabulary map.

```
<modal structure="Instance" vocabulary="Modal">
    <type name="Formula">
        <grammar>And,Or,Not,Box,Dia,Atom</grammar>
        <range>
            <lower>1</lower>
        </range>
    </type>
```

```
    <predicate name="And">
        <grammar>And</grammar>
        <arity>(Formula,Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Or">
        <grammar>Or</grammar>
        <arity>(Formula,Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Not">
        <grammar>Not</grammar>
        <arity>(Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Box">
        <grammar>Box</grammar>
        <arity>(Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Dia">
        <grammar>Dia</grammar>
        <arity>(Formula,Formula)</arity>
        <format>number_children</format>
    </predicate>
    <predicate name="Atom">
        <grammar>Atom</grammar>
        <arity>(Formula)</arity>
        <format>number</format>
        <terminal/>
    </predicate>
</modal>
```

C.5 Arithmetic Expressions

To add full arithmetic expressions to any suitable logic include the following grammar rules in the grammar of the logic.

```
Ex -> "(" Add ")" | "(" Sub ")" | "(" Mult ")" | "(" Div ")" |
       Var | Con
Add -> Ex "+" Ex
Sub -> Ex "-" Ex
Mult -> Ex "*" Ex
Div -> Ex "/" Ex
Var -> _character[lower]
Con -> _integer[IntMIN:IntMAX]
```

The following type and predicate tags should then be added to the vocabulary map.

```
<type name="Expression">
    <grammar>Add,Sub,Mult,Div,Var,Con</grammar>
    <range>
        <lower>1</lower>
    </range>
</type>
<type name="Number">
    <grammar></grammar>
    <range>
        <lower>IntMIN</lower>
        <upper>IntMAX</upper>
    </range>
    <inttype/>
</type>
<predicate name="Add">
    <grammar>Add</grammar>
    <arity>(Expression,Expression,Expression)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Subtract">
    <grammar>Sub</grammar>
    <arity>(Expression,Expression,Expression)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Multiply">
    <grammar>Mult</grammar>
    <arity>(Expression,Expression,Expression)</arity>
    <format>number_children</format>
</predicate>
<predicate name="Divide">
    <grammar>Div</grammar>
    <arity>(Expression,Expression,Expression)</arity>
    <format>number_children</format>
</predicate>
<predicate name="type">
    <grammar>Var</grammar>
    <arity>(Expression)</arity>
    <format>number</format>
    <terminal/>
</predicate>
<predicate name="Constant">
    <grammar>Con</grammar>
    <arity>(Expression,Number)</arity>
    <format>number_content</format>
    <terminal/>
```

```
</predicate>
```

Here we introduce a new keyword `number_content` within the format tag of predicate `Constant`. This keyword indicates that for nodes of type `Con` the arguments of its tuples are the identifier of the node and the content of the node. The content of a node is the value the node represents, only terminal nodes have content.

Appendix D

Supplementary Material

The accompanying zip file contains the INSTLATOR system. The zip file includes all files necessary to run both tools included in the INSTLATOR system as well as instructions and system requirements.

Filename: Instlator.zip