Aalto University
School of Science

Writing Declarative Specifications for Clauses

Martin Gebser[1,2], Tomi Janhunen[1], Roland Kaminski[2], Torsten Schaub[2,3], Shahab Tasharrofi[1]

[1)] Aalto University, Finland
[2)] University of Potsdam, Germany
[3)] INRIA Rennes, France

Computational Logic Day, Aalto University, December 8, 2015

Background: Boolean Satisfiability

Satisfiability (SAT) solvers provide an efficient implementation of classical propositional logic.

- SAT solvers expect their input in the conjunctive normal form (CNF), i.e., a conjunction of clauses $l_1 \vee \ldots \vee l_k$.
- Clauses can be viewed as "machine code" for expressing constraints and representing knowledge.
- Typically clauses are either
  - generated using a procedural program or
  - obtained when more complex formulas are translated.
- First-order formulas are prone to combinatorial effects:

$$\neg \mathrm{edge}(X, Y) \vee \neg \mathrm{edge}(Y, Z) \vee \neg \mathrm{edge}(Z, X) \vee \\ (X = Y) \vee (X = Z) \vee (Y = Z).$$

Analogue: Assembly Languages

```
smodels:
    pushq  %rbp
    movq  %rsp, %rbp
    subq  $32, %rsp
    movq  %rdi, -24(%rbp)
    movq  %rsi, -32(%rbp)
    movl  $0, -4(%rbp)
    movq  -32(%rbp), %rdx
    movq  -24(%rbp), %rax
    movq  %rdx, %rsi
    movq  %rax, %rdi
    call  propagate
    movq  %rax, -24(%rbp)
    movq  -24(%rbp), %rax
    movq  %rax, %rdi
    movl  $0, %eax
    call  conflict
    testl  %eax, %eax
    je   .L2
    movl  $0, %eax
    jmp   .L3
.L2:
    movq  -24(%rbp), %rax
    movq  %rax, %rdi
    movl  $0, %eax
    call  complete
    testl  %eax, %eax
    je   .L4
    movl  $-1, %eax
    jmp   .L3
    ...
.L3:
    leave
    ret
```

How to Generate Machine Code?

1. Assembly language
2. Assembly language + macros [tigcc.ticalc.org]

```
.macro  sum from=0, to=5
   .long   \from
   .if     \to-\from
   sum     "(\from+1)",\to
   .endif
   .endm
```

$\longmapsto$

```
    .long   0
    .long   1
    .long   2
    .long   3
    .long   4
    .long   5
```

3. High level language (C, C++, scala, ...) + compilation

How much can we control the actual output in each case?

Our Approach

- A fully declarative approach where intended clauses are given first-order specifications in analogy to ASP.
- In the implementation, we harness state-of-the-art ASP grounders for instantiating term variables in clauses.
- The benefits of our approach:
  - — Complex domain specifications supported
  - — Database operations available
  - — Uniform encodings enabled
  - — Elaboration tolerance
- WYSIWYG: $\mathrm{black}(X) \lor \mathrm{gray}(X) \lor \mathrm{white}(X) \leftarrow \mathrm{node}(X).$

$$\begin{array}{lcl} \mathrm{node}(a). & & \mathrm{black}(a)\lor\mathrm{gray}(a)\lor\mathrm{white}(a). \\ \mathrm{node}(b). & \longmapsto & \mathrm{black}(b)\lor\mathrm{gray}(b)\lor\mathrm{white}(b). \\ \mathrm{node}(c). & & \mathrm{black}(c)\lor\mathrm{gray}(c)\lor\mathrm{white}(c). \end{array}$$

Outline

Clause Programs

Modeling Methodology and Applications

Implementation

Discussion and Conclusion

Clause Programs: Syntax and Semantics

- The signature $\mathcal{P}$ for predicate symbols is partitioned into domain predicates $\mathcal{P}_{\mathrm{d}}$ and varying predicates $\mathcal{P}_{\mathrm{v}}$.
- Domain rules in $\mathcal{P}_{\mathrm{d}}$ are normal rules of the form

$$a \leftarrow c_1, \ldots, c_m, \sim d_1, \ldots, \sim d_n.$$

- The syntax for clause rules is

$$a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l \leftarrow c_1, \ldots, c_m, \sim d_1, \ldots, \sim d_n.$$

  where the head (resp. body) is expressed in $\mathcal{P}_{\mathrm{v}}$ (resp. $\mathcal{P}_{\mathrm{d}}$).

Definition

An Herbrand interpretation $I \subseteq \mathrm{Hb}(P)$ is a domain stable model of P iff $I \models \mathrm{Gnd}(P)$ and I_{d} is the least model of $\mathrm{Gnd}(P)^I$.

Example: Graph Coloring

Domain rules

$$\begin{array}{rcl} \mathrm{node}(X) & \leftarrow & \mathrm{edge}(X, Y). \\ \mathrm{node}(Y) & \leftarrow & \mathrm{edge}(X, Y). \end{array}$$

Clause rules

$$\begin{array}{rcl} \mathrm{black}(X) \vee \mathrm{gray}(X) \vee \mathrm{white}(X) & \leftarrow & \mathrm{node}(X). \\ \neg\mathrm{black}(X) \vee \neg\mathrm{black}(Y) & \leftarrow & \mathrm{edge}(X, Y). \\ \neg\mathrm{gray}(X) \vee \neg\mathrm{gray}(Y) & \leftarrow & \mathrm{edge}(X, Y). \\ \neg\mathrm{white}(X) \vee \neg\mathrm{white}(Y) & \leftarrow & \mathrm{edge}(X, Y). \end{array}$$

Uniform encoding that works for any graph instance!

Example: Continued

1. Suppose the following facts as instance information:

$$\text{edge}(a, b). \quad \text{edge}(b, c). \quad \text{edge}(c, a).$$

2. Additional domain atoms from $\text{node}(X; Y) \leftarrow \text{edge}(X, Y)$:

$$\text{node}(a), \quad \text{node}(b), \quad \text{node}(c).$$

3. Clauses from $\text{black}(X) \lor \text{gray}(X) \lor \text{white}(X) \leftarrow \text{node}(X)$:

$$\begin{array}{l} \text{black}(a) \lor \text{gray}(a) \lor \text{white}(a), \\ \text{black}(b) \lor \text{gray}(b) \lor \text{white}(b), \\ \text{black}(c) \lor \text{gray}(c) \lor \text{white}(c). \end{array}$$

4. Clauses from $\neg\text{black}(X) \lor \neg\text{black}(Y) \leftarrow \text{edge}(X, Y)$ etc:

$$\begin{array}{lll} \neg\text{black}(a) \lor \neg\text{black}(b), & \neg\text{gray}(a) \lor \neg\text{gray}(b), & \neg\text{white}(a) \lor \neg\text{white}(b), \\ \neg\text{black}(b) \lor \neg\text{black}(c), & \neg\text{gray}(b) \lor \neg\text{gray}(c), & \neg\text{white}(b) \lor \neg\text{white}(c), \\ \neg\text{black}(c) \lor \neg\text{black}(a), & \neg\text{gray}(c) \lor \neg\text{gray}(a), & \neg\text{white}(c) \lor \neg\text{white}(a). \end{array}$$

Encodings

We have published some illustrative encodings:

1. Graph *n*-Coloring
2. *n*-Queens
3. Markov Network Structure Learning
4. Instruction Scheduling

The encodings are available at
`http://research.ics.aalto.fi/software/sat/satgrnd/`

More details can be found from our GTTV'15 paper:
`http://research.ics.aalto.fi/software/sat/etc/satgrnd.pdf`

Graph n-Coloring

- The number of colors is parametrized by n.
- We may exploit many advanced features of the grounder:
  - Range specifications
  - Pooling
  - Conditional literals
- If need be, the lengths of clauses can vary dynamically depending on the problem instance!

$$
\begin{array}{rcl}
\mathrm{color}(1 \ldots n). & & \\
\mathrm{node}(X; Y) & \leftarrow & \mathrm{edge}(X, Y). \\
\bigvee \mathrm{hascolor}(X, C) : \mathrm{color}(C) & \leftarrow & \mathrm{node}(X). \\
\neg\mathrm{hascolor}(X, C) \vee \neg\mathrm{hascolor}(Y, C) & \leftarrow & \mathrm{edge}(X, Y), \mathrm{color}(C).
\end{array}
$$

More Complex Domains: Highlights

1. *n*-Queens:

$$\text{attack}(X+R, Y+C, R, C) \vee \neg\text{attack}(X, Y, R, C) \leftarrow \text{target}(X, Y, R, C), \text{target}(X-R, Y-C, R, C).$$

2. Markov Network Structure Learning:

$$\text{del}(X, L) \vee \bigvee \text{edge}(X_1, Y_1, L) : \text{maps}(X, Y, X_1, Y_1) : Y \neq Z \leftarrow \text{node}(X), \text{node}(Z), X \neq Z, \text{level}(L).$$

3. Instruction Scheduling (MaxSAT):

$$\text{value}(I, S) \vee \bigvee \neg\text{delay}(I, S) : \text{range}(I, S-1) \vee \bigvee \text{delay}(I, S+1) : \text{range}(I, S+1) \leftarrow \text{range}(I, S).$$

Tool Support

- An adapter called SATGRND can be used to transform the output of standard GRINGO (v. 2 onward) into DIMACS.
- If optimization statements are used, a (weighted partial) MaxSAT instance will be produced in DIMACS format.
- Enhanced user experience enabled by symbolic names:

```
gringo myprog.lp | satgrnd \
| owbo-acycglucose -print-method=1 -verbosity=0
```

- The required tools are available for download as follows:

  GRINGO: potassco.sourceforge.net/
  SATGRND and OWBO-ACYCGLUCOSE:
  research.ics.aalto.fi/software/sat/download

Related Work

- Procedural approaches: Python interface of Microsoft's Z3.
- Declarative approaches:
  - Propositional schemata and PSGRND [East et al., 2006]
  - Grounding first-order formulas [Aavani et al., 2011; Blockeel et al., 2012; Jansen et al., 2014; Wittocx et al., 2010]
  - IDP3 [Jansen et al., 2013]
  - Datalog in planning domains [Helmert, 2009]
  - Translating constraint models into CNF [Huang, 2008]
- Strengths combined by the GRINGO interface:
  - Domains definable using rules in a declarative way
  - Recursive domain definitions supported
  - Domains not finitely bounded a priori
  - General-purpose grounder

Conclusion

- In this work, we suggest to write declarative first-order specifications (with term variables) for clauses.
- Advantages of using an ASP grounder for instantiation:
  - Exact clause-level control over the output
  - All advanced features of the grounder available
  - Uniform encodings enabled
  - Elaboration tolerance
- The combination of GRINGO and SATGRND provides a general-purpose grounder for SAT and MaxSAT.
- Further extensions to SATGRND are being developed:
  - Support for acyclicity constraints
  - Back-end translator to other formats (SMTLIB, MIP, PB)

Beyond Clauses

Propositional logic can be implemented using SATGRND:

1. Domains of subsentences, compounds, and atoms:

$$\begin{array}{ll}
\mathrm{sub}(S) \leftarrow \mathrm{sat}(S). & \\
\mathrm{sub}(S1; S2) \leftarrow \mathrm{sub}(a(S1, S2)). & \mathrm{co}(a(S1, S2)) \leftarrow \mathrm{sub}(a(S1, S2)). \\
\mathrm{sub}(S1; S2) \leftarrow \mathrm{sub}(o(S1, S2)). & \mathrm{co}(o(S1, S2)) \leftarrow \mathrm{sub}(o(S1, S2)). \\
\mathrm{sub}(S) \leftarrow \mathrm{sub}(n(S)). & \mathrm{co}(n(S)) \leftarrow \mathrm{sub}(n(S)). \\
\mathrm{true}(S) \leftarrow \mathrm{sat}(S). & \mathrm{at}(S) \leftarrow \mathrm{sub}(S), \sim\mathrm{co}(S).
\end{array}$$

2. Tseitin transformations (e.g., for $a(S1, S2)$):

$$\begin{array}{l}
\mathrm{true}(a(S1, S2)) \vee \neg\mathrm{true}(S1) \vee \neg\mathrm{true}(S2) \leftarrow \mathrm{co}(a(S1, S2)). \\
\neg\mathrm{true}(a(S1, S2)) \vee \mathrm{true}(S1) \leftarrow \mathrm{co}(a(S1, S2)). \\
\neg\mathrm{true}(a(S1, S2)) \vee \mathrm{true}(S2) \leftarrow \mathrm{co}(a(S1, S2)).
\end{array}$$

3. Sentences to satisfy as instance information:

$$\mathrm{sat}(o(n(a), b)). \quad \mathrm{sat}(o(n(b), c)). \quad \mathrm{sat}(o(n(c), a)).$$