Aalto University
School of Science

Writing Declarative Specifications for Clauses

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

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

JELIA’16, Larnaca, Cyprus, November 10, 2016

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:

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

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 terms variables in clauses.
- The benefits of our approach:
  - Complex domain specifications supported
  - Database operations available
  - Uniform encodings enabled
  - Elaboration tolerance
- WYSIWYG: $\text{black}(X) \lor \text{gray}(X) \lor \text{white}(X) \leftarrow \text{node}(X).$

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

Outline

Clause Programs

Modeling Methodology and Applications

Implementation

Discussion and Conclusion

Clause Programs: Syntax

- The signature $\mathcal{P}$ for predicate symbols is partitioned into domain predicates $\mathcal{P}_{\text{d}}$ and varying predicates $\mathcal{P}_{\text{v}}$.
- Domain rules in $\mathcal{P}_{\text{d}}$ 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 $\mathcal{P}_{\text{v}}$ (resp. $\mathcal{P}_{\text{d}}$).
- For standard use cases, the domain part of a program should be stratified to enable evaluation by the grounder.

Example: Graph Coloring

Domain rules

$$\mathrm{node}(X) \leftarrow \mathrm{edge}(X, Y).$$
$$\mathrm{node}(Y) \leftarrow \mathrm{edge}(X, Y).$$

Clause rules

$$\mathrm{black}(X) \vee \mathrm{gray}(X) \vee \mathrm{white}(X) \leftarrow \mathrm{node}(X).$$
$$\neg\mathrm{black}(X) \vee \neg\mathrm{black}(Y) \leftarrow \mathrm{edge}(X, Y).$$
$$\neg\mathrm{gray}(X) \vee \neg\mathrm{gray}(Y) \leftarrow \mathrm{edge}(X, Y).$$
$$\neg\mathrm{white}(X) \vee \neg\mathrm{white}(Y) \leftarrow \mathrm{edge}(X, Y).$$

Uniform encoding that works for any graph instance!

Clause Programs: Semantics

- The Herbrand universe $\mathrm{Hu}(P)$ and the Herbrand base $\mathrm{Hb}(P)$ of a clause program P are defined as usual.
- The ground program $\mathrm{Gnd}(P)$ is the respective Herbrand instantiation of P over the universe $\mathrm{Hu}(P)$.
- The domain reduct P^I of P with respect to I contains the positive rule $a \leftarrow c_1, \ldots, c_m$ for every domain rule

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

such that $\{d_1, \ldots, d_n\} \cap I_\mathrm{d} = \emptyset$.

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: Continued

1. Suppose the following facts as instance information:

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

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

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

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

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

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

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

Encodings

In the paper, we illustrate the use of clause programs:

- Graph *n*-coloring
- *n*-Queens
- Full propositional logic
- Haplotype inference

Further sample encodings can found from our website:

- Structure learning for Markov networks
- Instruction scheduling
- SuDoku puzzles

`http://research.ics.aalto.fi/software/sat/satgrnd/`

Graph *n*-Coloring

- The number of colors is parameterized by *n*.
- We may exploit many advanced features of the grounder:
  - Range specifications
  - Pooling (substantially revised in GRINGO v. 4)
  - Conditional literals
- If need be, the lengths of clauses can vary dynamically depending on the problem instance!

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

More Complex Domains: Highlights

- Parameterization and non-trivial domains in n-Queens:

$$\text{coord}(1 \ldots n).$$
$$\text{dir}(0, -1). \quad \text{dir}(-1, 0). \quad \text{dir}(-1, -1). \quad \text{dir}(-1, 1).$$
$$\text{target}(X, Y, R, C) \leftarrow \text{coord}(X; Y; X+R; Y+C), \text{dir}(R, C).$$
$$\text{attack}(X+R, Y+C, R, C) \vee \neg\text{attack}(X, Y, R, C) \leftarrow$$
$$\text{target}(X, Y, R, C), \text{target}(X-R, Y-C, R, C).$$

- Dynamic-length clauses for haplotype inference:

$$\text{used}(G_2, E_2) \vee$$
$$\bigvee \text{same}(G_1, E_1, G_2, E_2) :$$
$$(\text{keep}(G_1),\ E_1 = 0 \ldots 1,\ (G_1, E_1) < (G_2, E_2)) \leftarrow$$
$$\text{keep}(G_2),\ E_2 = 0 \ldots 1.$$

Beyond Clauses: Full Propositional Logic

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 $a(S1, S2)$):

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

3. Sentences to satisfy as instance information:

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

Implementation Strategy

- Clause programs can be directly grounded using the state-of-the-art ASP grounder GRINGO (v. 2 onward).
- The output of GRINGO is a ground disjunctive logic program $\mathrm{Gnd}(P)$ consisting of bodyless disjunctive rules

  $$a_1 \vee \cdots \vee a_k \vee \neg b_1 \vee \cdots \vee \neg b_l.$$

  where each a_i and $\neg b_j$ is a classical literal over $\mathrm{Hb_v}(P)$.
- The semantic connection of a and its negation $\neg a$ can be re-established by viewing such disjunctions as clauses.
- The classical models of $\mathrm{Gnd}(P)$ correspond to the domain stable models of the clause program P.

Tool Support

- An adapter called SATGRND can be used to transform the output of GRINGO 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 at

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

Haplotype Inference Re-Engineered

- The RPOLY system is a reference implementation of haplotype inference [Graça et al., 2007] based on
  - procedurally generated PB optimization instances and
  - the use of MINISAT+ as the back-end PB solver.
- The performance improved 40× by remodeling the problem with SATGRND and using CLASP as the PB solver.

	RPOLY	$\Rightarrow$	$\Rightarrow$-LB	$\Leftrightarrow$	$\Leftrightarrow$-LB	
t	182.3	**3.3**	3.5	4.7	5.5	CLASP
$\lvert\rightleftharpoons\rvert$	466,933	47,262	52,420	57,789	67,178	
$\lvert C\rvert$	36,299	28,318	28,454	49,054	49,192	
t	**133.6**	1789.8	1402.7	2639.1	2467.4	MINISAT+
$\lvert\rightleftharpoons\rvert$	863,514	6,779,058	6,441,567	6,769,964	5,866,433	
$\lvert C\rvert$	36,859	28,500	28,638	51,003	51,142	

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
  - 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.
- Other back-end formats are supported: SMT, MIP, PB.
- Further extensions to SATGRND are being developed:
  - — Support for acyclicity constraints