Satisfiability and Preferences: Theory and Applications to Planning and Testing

Enrico Giunchiglia

Laboratory of Systems and Technologies for Automated Reasoning (STAR-Lab)
DIST - Univ. Genova
Thanks to: E. Di Rosa, M. Maratea, P. Marin, M. Narizzano, A. Puddu . . .

4th International Seminar on New Issues in Artificial Intelligence
Madrid, Jan. 31st - Feb. 4th 2011

Motivations - 1

Propositional Satisfiability (SAT) is a success story in CS/AI:

1. current SAT solvers can determine the satisfiability or unsatisfiability of problems with hundreds of thousands variables, and
2. to solve a combinatorial problem P it is often better to translate it to SAT and then apply a SAT solver than using a dedicate solver for P.

Motivations - 2

However,

1. In many cases it is not sufficient to determine the satisfiability of a set of constraints,
2. E.g., we may want to find models satisfying also as many other constraints as possible (as in MAXSAT) or which minimize a given objective function (as in MINONE),
3. These additional preferences, are often independent of the set of constraints and can be partially ranked qualitatively or quantitatively, and
4. introduce a ranking on the models and thus the notion of optimal model.

Goals of the talk

1. Introduce a simple formalism for expressing (qualitative) preferences
2. Show how it is possible to compute optimal models by slightly modifying existing SAT solvers
3. discuss the applicability of the theory to Planning (done) and ATG (work in progress).

Outline

Outline

1. Propositional Satisfiability (SAT)
2. Preferences
   - Computing one/all optimal solution(s) by guiding search
   - Computing one/all optimal solution(s) by generate&test
   - Experimental results
3. Planning as satisfiability
   - Automated Symbolic Planning
   - Soft Goals
   - Plan quality
4. Automatic Test Generation
   - Automatic Test Generation with CBMC
   - Experimental Analysis

Boolean logic: syntax

Given a set of variables $X = \{x_1, \ldots, x_n\}$ a formula is either:

- a variable $x \in X$
- $\neg\alpha$ where α is a formula
- $(\alpha \circ \beta)$ where α, β are formulas and $\circ \in \{\wedge, \vee, \supset, \equiv, \oplus\}$

Example

$(x_1 \wedge \neg x_1)$	Absurdity
$(\neg(x_1 \vee x_2) \equiv (\neg x_1 \wedge \neg x_2))$	De Morgan's law
$((x_1 \oplus x_3) \oplus (x_2 \wedge x_4))$	MSB of a 2-bit adder with inputs $x_1 x_2$ and $x_3 x_4$

Propositional Satisfiability

Formula

- A formula φ is a set $\{C_1, \ldots, C_n\}$ of clauses
- A clause C_i is a set $\{l_1, \ldots, l_m\}$ of literals
- A literal l is a variable x or the negation $\overline{x}$ of a variable x.

Assignment

- An assignment μ is a maximally consistent set of literals
- An assignment μ satisfies (or is a model of) a formula φ if for each $C \in \varphi$, $\mu \cap C \neq \emptyset$

Propositional Satisfiability

Arbitrary formulas (not necessarily in CNF) and mathematical (linear) expressions with finite range variables can be coded in SAT.

$$s_1 = (\overbrace{(x_1 \oplus x_3)}^{s} \oplus \underbrace{(x_2 \wedge x_4)}_{c})$$

(s_1 is the MSB of a 2 bit adder with input $x_1 x_2$ and $x_3 x_4$)

$$\begin{array}{l}(s \vee c)\wedge \\ (\neg s \vee \neg c)\wedge \\ (\neg s \vee x_1 \vee x_3)\wedge \\ (\neg s \vee \neg x_1 \vee \neg x_3)\wedge \\ (s \vee \neg x_1 \vee x_3)\wedge \\ (s \vee x_1 \vee \neg x_3)\wedge \\ (c \vee \neg x_2 \vee \neg x_4)\wedge \\ (\neg c \vee x_2)\wedge \\ (\neg c \vee x_4)\end{array}$$

$$\begin{array}{l}\{ \\ \quad \{s, c\}, \\ \quad \{\overline{s}, \overline{c}\}, \\ \quad \{\overline{s}, x_1, x_3\}, \\ \quad \{\overline{s}, \overline{x}_1, \overline{x}_3\}, \\ \quad \{s, \overline{x}_1, x_3\}, \\ \quad \{s, x_1, \overline{x}_3\}, \\ \quad \{c, \overline{x}_2, \overline{x}_4\}, \\ \quad \{\overline{c}, x_2\}, \\ \quad \{\overline{c}, x_4\} \\ \}\end{array}$$

$\{l\}$ is a unit clause

Search algorithm

DLL(φ,μ)
1 **if** all clauses are satisfied
    then return μ
2 **if** a clause is violated
    then return FALSE
3 **if** $\{l\}$ is a unit clause in φ
    then return DLL(φ_l,$\mu \cup \{l\}$)
4 l := a literal in φ;
    return DLL(φ_l,$\mu \cup \{l\}$) **or**
        DLL($\varphi_{\bar{l}}$,$\mu \cup \{\bar{l}\}$)

[Davis, Logemann, Loveland 1962]

Key technologies in today's DLL implementations:

1. (Lazy) data structures for efficient unit clause detection
2. (UIP-based) Learning for backjumping over irrelevant nodes and avoid repeating the same mistakes in different parts of the search tree
3. Dynamic heuristics either based on learning or on unit-propagation

minisat has less than 1Kloc

Search algorithm

DLL(φ,μ)

1 **if** all clauses are satisfied
   then return μ

2 **if** a clause is violated
   then return FALSE

3 **if** $\{l\}$ is a unit clause in φ
   then return DLL(φ_l,$\mu \cup \{l\}$)

4 l := a literal in φ;
   return DLL(φ_l,$\mu \cup \{l\}$) **or**
   DLL($\varphi_{\bar{l}}$,$\mu \cup \{\bar{l}\}$)

[Davis, Logemann, Loveland 1962]

Key technologies in today's DLL implementations:

1. (Lazy) data structures for efficient unit clause detection
2. (UIP-based) Learning for backjumping over irrelevant nodes and avoid repeating the same mistakes in different parts of the search tree
3. Dynamic heuristics either based on learning or on unit-propagation

minisat has less than 1Kloc

Example

Assume we are given $\{\overline{Fish}, \overline{Meat}\}$, DLL will return

1. $\{\overline{Fish}, \overline{Meat}\}$ if DLL branches on $\{\overline{Fish}, \overline{Meat}\}$,
2. $\{Fish, \overline{Meat}\}$ if DLL branches on $\{Fish\}$,
3. $\{\overline{Fish}, Meat\}$ if DLL branches on $\{Meat\}$.

However, it may be the case that not all models are equally good. For instance, I may prefer to stay on diet, or I may prefer *Fish* to *Meat*, or viceversa.

Outline

Qualitative Preferences on Literals - 1

Qualitative Preference on Literals

A preference is a poset $S, \prec$ of literals.

Intuitively:

1. S represents the literals that we would like to have satisfied,
2. $\prec$ models the relative importance of our preference.

Example

$S = \{Fish, Meat\}$ with $Fish \prec Meat$, models the fact we like to have *Fish* and *Meat* and that we prefer *Fish* over *Meat*.

Qualitative Preferences on Literals - 2

$\mu \prec \mu'$

μ and μ' are assignments. $\mu \prec \mu'$ if and only if

1. there exists a literal $l \in S$ with $l \in \mu$ and $\bar{l} \in \mu'$; and
2. $\forall l \in S \cap (\mu' \setminus \mu), \exists l' \in S \cap (\mu \setminus \mu')$ such that $l' \prec l$.

Example

$S = \{Fish, Meat\}$ with $Fish \prec Meat$. Then,

$$\{Fish, Meat\} \prec \{Fish, \overline{Meat}\} \prec \{\overline{Fish}, Meat\} \prec \{\overline{Fish}, \overline{Meat}\};$$

If φ is $(\overline{Fish} \vee \overline{Meat})$, the optimal model is $\{Fish, \overline{Meat}\}$.

Qualitative Preferences on Literals - 2

Advantages:

1. Simple,
2. Can be generalized to "qualitative preferences on formulas" by introducing "definitions" or "names"
3. Can be used to model "quantitative preferences on literals/formulas" by Boolean encoding of the objective function

Quantitative preferences

1. $\langle S, c \rangle$ is a quantitative preference, where S is a set of literals (formulas) and $c : S \mapsto N$.
2. The cost of an assignment μ is

$$\sum_{l \in S : \mu \not\models l} c(l).$$

3. A model (of a given formula) is optimal if it has a minimal associated cost.

From quantitative to qualitative preferences

1. φ is the given formula, and S, c is a quantitative precerence
2. $adder(S, c)$ is a Boolean formula corresponding to the cost function
3. $b_m, \ldots, b_0$ is the sequence of variables (bits) representing the value of $adder(S, c)$
4. S' is the set $\{\overline{b}_m, \ldots, \overline{b}_0\}$ with
5. $\overline{b}_m \prec' \overline{b}_{m-1} \prec' \ldots \prec' \overline{b}_0$

Fact

A model μ is optimal wrt S, c iff μ is optimal wrt $S', \prec'$.

Indeed, if our preferences are such that

$$\overline{b}_m \prec' \overline{b}_{m-1} \prec' \ldots \prec' \overline{b}_0$$

then, the set of assignments is ordered as follows:

$$\begin{aligned}
&\overline{b}_m\overline{b}_{m-1}\ldots\overline{b}_0\ldots \\
&\qquad \prec' \overline{b}_m\overline{b}_{m-1}\ldots b_0\ldots \\
&\qquad\qquad \prec' \ldots \\
&\qquad\qquad\qquad \prec' b_m b_{m-1}\ldots b_0\ldots.
\end{aligned}$$

For instance, if $m = 1$, we have

$$\{\overline{b}_1, \overline{b}_0 \ldots\} \prec' \{\overline{b}_1, b_0 \ldots\} \prec' \{b_1, \overline{b}_0 \ldots\} \prec' \{b_1, b_0 \ldots\}.$$

There are several ways to compute $adder(S, c)$:

Warners 1998 : linear number of clauses ($8|S|$) and variables ($2|S|$)

Bailleux & Boufkhad 2003 : quadratic number of clauses

Sinz 2005 : (slightly) improved versions of the previous

OPTSAT

$S, \prec$:= a qualitative preference on literals;
$\psi := \top$; $\mu_{opt} := \emptyset$

function OPTSAT-R(φ,μ)

1 **if** $(\bot \in (\varphi \cup \psi)_\mu)$ **return** FALSE;
2 **if** (μ is total) $\mu_{opt} := \mu$; $\psi := Reason(\mu, S, \prec)$; **return** FALSE;
3 **if** $(\{l\} \in (\varphi \cup \psi)_\mu)$ **return** OPTSAT-R$(\varphi, \mu \cup \{l\})$;
4 $l := ChooseLiteral(\varphi \cup \psi, \mu)$;
5 **return** OPTSAT-R$(\varphi, \mu \cup \{l\})$ **or** OPTSAT-R$(\varphi, \mu \cup \{\bar{l}\})$.

Fact

At the end of computation, μ_{opt} stores an optimal model if φ is satisfiable, and the empty set otherwise.

Outline

One optimal solution by guiding search (ECAI'06)

Given a set of clauses φ and a qualitative prefence $S, \prec$, an optimal model can be computed by

1. modifying DLL heuristics in order to branch following the partial order on literals:

   Pros : The first generated model is guaranteed to be optimal

   Cons : Imposing an order on the heuristic can produce an exponential degradation in the performances (at least in theory).

Example

Assumptions

1. $S = \{Fish, Meat\}$ with $Fish \prec Meat$

In this case, OPTSAT-06 looks for models with

1. $\{Fish, Meat\}$. If no such model exists
2. $\{Fish, \overline{Meat}\}$. If no such model exists
3. $\{\overline{Fish}, Meat\}$. If no such model exists
4. $\{\overline{Fish}, \overline{Meat}\}$. If no such model exists
5. returns False.

If φ is $(\overline{Fish} \vee \overline{Meat})$ the model returned is $\{Fish, \overline{Meat}\}$.

All optimal solutions by guiding search (CP'08)

Given a set of clauses φ and a qualitative prefence $S, \prec$, all optimal models can be computed by

1. modifying DLL heuristics in order to branch following the partial order on literals;
2. When an optimal model μ is found, a formula ψ blocking the models dominated by μ is added to φ;
3. search is continued looking for other optimal models.

Pros : ψ is computed in polynomial time

Pros : Only optimal models are generated

Cons : Whenever an optimal model is generated, a formula is added to the input one.

Outline

One optimal solution by generate&test (ECAI'08)

Given a set of clauses φ and a qualitative prefence $S, \prec$, an optimal model can be computed by

1. generating a not necessarily optimal model μ of φ,
2. testing if μ is optimal and
   1. returning μ if optimal, or
   2. adding a formula which forces models $\mu' \prec \mu$ and continuing the search otherwise.

Pros : No modifications in the heuristic is needed

Cons : It may generate non-optimal models.

Solving SAT problems with preferences - II

Fact

Let μ and μ' be two total assignments. Let $S, \prec$ be a qualitative preference. μ' is preferred to μ wrt $S, \prec$ if and only if μ' satisfies

$$(\vee_{l:l\in S, l\notin\mu} l) \wedge (\wedge_{l':l'\in S, l'\in\mu}(\vee_{l:l\in S, l\notin\mu, l\prec l'} l \vee l')). \tag{1}$$

Special cases:

- if $S = \emptyset$ or $S \subseteq \mu$, (1) is FALSE, i.e., μ is already optimal;
- if $\prec = \emptyset$, (1) is

$$(\vee_{l:l\in S, l\notin\mu} l) \wedge (\wedge_{l:l\in S, l\in\mu} l),$$

i.e., for any assignment μ' satisfying (1),

$$S \cap \mu \subset S \cap \mu'$$

Solving SAT problems with preferences - II

Fact

Let μ and μ' be two total assignments. Let $S, \prec$ be a qualitative preference. μ' is preferred to μ wrt $S, \prec$ if and only if μ' satisfies

$$(\vee_{l:l \in S, l \notin \mu} l) \wedge (\wedge_{l':l' \in S, l' \in \mu} (\vee_{l:l \in S, l \notin \mu, l \prec l'} l \vee l')). \qquad (1)$$

Special cases:

- if $S = \emptyset$ or $S \subseteq \mu$, (1) is FALSE, i.e., μ is already optimal;
- if $\prec = \emptyset$, (1) is

$$(\vee_{l:l \in S, l \notin \mu} l) \wedge (\wedge_{l:l \in S, l \in \mu} l),$$

i.e., for any assignment μ' satisfying (1),

$$S \cap \mu \subset S \cap \mu'$$

Example

Assumptions

$\varphi = (\overline{Fish} \vee \overline{Meat})$, $S = \{Fish, Meat\}$ and $Fish \prec Meat$

In the "worst" case, OPTSAT-08

1. finds the model $\{\overline{Fish}, \overline{Meat}\}$, and then looks for models of $(\overline{Fish} \vee \overline{Meat}) \wedge (Fish \vee Meat)$;
2. finds the model $\{\overline{Fish}, Meat\}$, and then looks for models of $(\overline{Fish} \vee \overline{Meat}) \wedge Fish$;
3. finds the model $\{Fish, \overline{Meat}\}$, and then looks for models of $(\overline{Fish} \vee \overline{Meat}) \wedge Fish \wedge Meat$;
4. returns FALSE: the last model found is optimal.

All optimal solutions by generate&test

Given a set of clauses φ and a qualitative prefence $S, \prec$, all optimal models can be computed by

1. generating a not necessarily optimal model μ of φ,
2. testing if μ is optimal and
    1. printing μ if optimal,
    2. learning a formula blocking the assignments $\mu' : \mu \prec \mu'$, and continuing the search.

Pros : No modifications in the heuristic is needed

Cons : It may generate non-optimal models.

Outline

MIN-ONE and MIN-ONE$_\subseteq$

Setting

φ is a formula; P is the set of variables in φ; μ, μ' total assignments.

MIN-ONE and MIN-ONE$_\subseteq$

The goal is to minimize the set of variables set to TRUE. Two possible definitions:

1. Quantitative, MIN-ONE: $\mu \prec \mu'$ if $|\mu \cap P| < |\mu' \cap P|$.
2. Qualitative, MIN-ONE$_\subseteq$: $\mu \prec \mu'$ if $\mu \cap P \subset \mu' \cap P$.

Solving MIN-ONE and MIN-ONE$_{\subseteq}$ with preferences

Setting

φ is a formula; P is the set of variables in φ; $\overline{P} = \{\overline{x} : x \in P\}$

MIN-ONE$_{\subseteq}$

μ is an optimal model of MIN-ONE$_{\subseteq}$ iff μ is an optimal model wrt the qualitative preference $\overline{P}, \emptyset$.

MIN-ONE

μ is an optimal model of MIN-ONE iff μ is an optimal model wrt the quantitative preference $\overline{P}, c$ with c a constant function.

MAX-SAT and MAX-SAT$_\subseteq$

Setting

φ is a formula; μ, μ' are total assignments.

MAX-SAT and MAX-SAT$_\subseteq$

1. In MAX-SAT$_\subseteq$ problems, $\mu \prec \mu'$ if the set of clauses satisfied by μ is a superset of the clauses satisfied by μ'
2. In MAX-SAT problems, $\mu \prec \mu'$ if the cardinality of the set of clauses satisfied by μ is bigger than the cardinality of the set of the clauses satisfied by μ'

From MAX-SAT and MAX-SAT$_\subseteq$ to MIN-ONE and MIN-ONE$_\subseteq$

Intuition

1. Given a formula φ, a new variable v_i is added to the clause $C_i \in \varphi$, e.g., $\{\{a\},\{b,c\}\}$ becomes $\{\{a,v_1\}\{b,c,v_2\}\}$.
2. MAX-SAT and MAX-SAT$_\subseteq$ correspond to miniminize the set of variables v_i assigned to 1, and thus reduce (with some care) to MIN-ONE and MIN-ONE$_\subseteq$ problems.

Experimental results

	class	n	OPTSAT-06	OPTSAT-08	OPTSAT-06	OPTSAT-08
1	Partial MINONE	21	77.99(19)	2.7(21)	74.28(21)	69.89(21)
2	MINONE	26	0.69(26)	0.2(26)	93.24(24)	23.99(25)
3	MAXSAT	35	26.68(34)	11.25(35)	218.86(31)	175.12(31)
4	MAXCUT/spinglass	5	0.01(5)	0.01(5)	7.56(1)	7.52(1)
5	MAXCUT/dimacs_mod	62	0.01(62)	0.01(62)	66.86(4)	21.61(3)
6	PSEUDO/garden	7	0.02(7)	0.01(7)	22.8(5)	36.66(5)
7	PSEUDO/logic-synthesis	17	0.03(17)	0.01(17)	90.36(3)	338.26(3)
8	PSEUDO/primes	148	4.81(130)	0.19(131)	31.8(103)	60.59(109)
9	PSEUDO/routing	15	11.69(15)	3.12(15)	41.49(15)	36.1(15)
10	MAXONE/structured	60	0.96(60)	0.13(60)	293(56)	7.87(58)
11	MAXCLIQUE/structured	62	0.01(62)	0.06(62)	54.14(19)	178.04(23)

Table: Qualitative (cols 4-5) and Quantitative (cols 6-7) preferences

Experimental results: considerations

1. MIN-ONE/MIN-ONE$_\subseteq$ and MAX-SAT/MAX-SAT$_\subseteq$ problems involve large set of objects (the whole set of variables and clauses respectively in the formula).
2. In many application domains, like planning, the optimization problem involves a few objects, and in these cases we can expect OPTSAT-06 to perform well
3. The results for the quantitative case can vary depending on the way the objective function is encoded (e.g, Warners, Boufkhad,. . .)
4. The results for OPTSAT-08 depend on the number of intermediate models generated before finding the optimal

Experimental analysis - OPTSAT-06

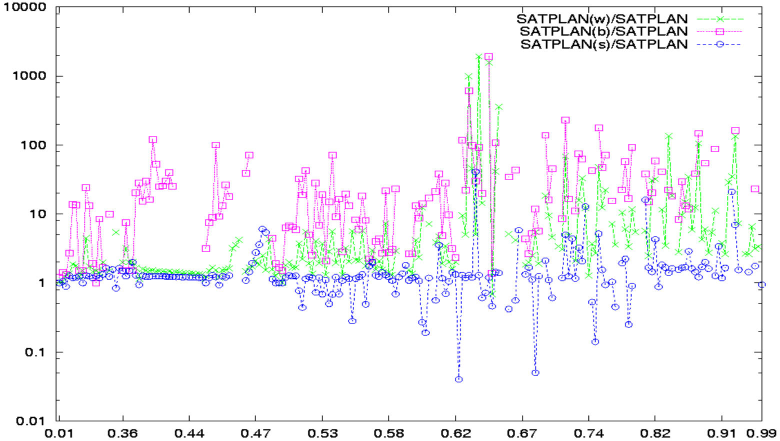


Figure: Performances of SATPLAN(w)/(b)/(s) wrt SATPLAN as a

Experimental results - OPTSAT-08

	class	T_1	Q_1	nSols	T_f	Q_f
1	Partial MINONE	2.68	45.5	1.5	2.7	44.1
2	MINONE	0.19	751.6	1	0.2	751.6
3	MAXSAT	0.05	8605.2	20.2	11.25	8847.6
4	MAXCUT/spinglass	0.01	770.4	1	0.01	770.4
5	MAXCUT/dimacs_mod	0.01	695.9	1.2	0.01	701.9
6	PSEUDO/garden	0.01	496	1	0.01	496
7	PSEUDO/logic-synthesis	0.01	152.2	1	0.01	152.2
8	PSEUDO/primes	0.18	368.4	1	0.19	368.4
9	PSEUDO/routing	3.12	58.7	1	3.12	58.7
10	MAXONE/structured	0.12	240.5	7.4	0.13	249.8
11	MAXCLIQUE/structured	0.06	430.4	1	0.06	430.4

Table: CPU time and Quality for finding first (columns T_1/Q_1) and optimal (columns T_f/Q_f) solution. number of models computed (column nSols).

Outline

Outline

Planning Problem

States & actions

Let F and A be the set of fluents and actions resp.

- A state is an assignment over the fluent signature
- An action is an assignment over the action signature

Planning problem

A planning problem is a triple $\langle I, tr, G \rangle$ where

- I is a formula over F: the set of initial states
- tr is a formula over $F \cup A \cup F'$: the transition relation
- G is a Boolean formula over F: the set of goal states

The classical problem is: Is there a sequence of transitions leading from I to G?

Planning Problem

States & actions

Let F and A be the set of fluents and actions resp.

- A state is an assignment over the fluent signature
- An action is an assignment over the action signature

Planning problem

A planning problem is a triple $\langle I, tr, G\rangle$ where

- I is a formula over F: the set of initial states
- tr is a formula over $F \cup A \cup F'$: the transition relation
- G is a Boolean formula over F: the set of goal states

The classical problem is: Is there a sequence of transitions leading from I to G?

Assumptions

1. Propositional setting: no metric quantities (resources, duration)
2. A planning problem can be expressed in your favourite formalism (STRIPS, ADL, $\mathcal{C}$, . . .)
3. A planning problem corresponds to a symbolic reachability problem

Example: STRIPS

In STRIPS, $\langle I, tr, G\rangle$ is such that:

- I is specified as a set of fluents I_S:

$$I = \wedge_{F \in I_S} F \wedge \wedge_{F \notin I_S} \neg F$$

- G is specified as a set of fluents G_S:

$$G = \wedge_{F \in G_S} F \wedge \wedge_{F \notin G_S} \neg F$$

- tr is specified by giving, for each action a, the three sets $Pre(a)$, $Add(a)$ and $Del(a)$

STRIPS Example

Drive(SL,GE):
- Pre: At(SL)
- Add: At(GE)
- Del: At(SL)

Fly(GE,MI):
- Pre: At(GE)
- Add: At(MI)
- Del: At(GE)

Drive(GE,MI):
- Pre: At(GE)
- Add: At(MI)
- Del: At(GE)

Encoding STRIPS descriptions in SAT

Let i be an integer.

1. If an action is executed at time i, its preconditions are satisfied, e.g.,

$$Drive_i(SL, GE) \supset At_i(SL)$$

2. If an action is executed at time i, its effects hold at time $i+1$, e.g.,

$$Drive_i(SL, GE) \supset \neg At_{i+1}(SL) \wedge At_{i+1}(GE)$$

3. Changes occur only because of action, e.g.,

$$At_{i+1}(MI) \wedge \neg At_i(MI) \supset Fly_i(GE, MI) \vee Drive_i(GE, MI)$$

4. It is not possible to execute two conflicting actions at the same time, e.g.,

$$\neg(Drive_i(SL, GE) \wedge Drive_i(GE, MI))$$

Classical Planning as Satisfiability

Assumptions

1. $\langle I, tr, G\rangle$ is a deterministic planning problem:
    1. there is only one state satisfying I
    2. for each state s and complex action a there is at most one state s' satisfying tr

Planning problem (of length n)

A planning problem (of length n) is the Boolean formula

$$I_0 \wedge \wedge_{i=1}^{n} tr_i \wedge G_n$$

Plan (of length n)

Given a planning problem φ of length n, a plan (of length n) is an assignment satisfying φ.

Classical Planning with (hard) Constraints

Example (State (Qualification) Constraints)

It is not possible to be in two locations at the same time. Just add to φ:

$$\wedge_{i=0}^{n} \neg(At_i(GE) \wedge At_i(SL))$$

Example (Trajectory Constraints on Action Variables)

I do want to drive after flying. Just add to φ:

$$\wedge_{i=0}^{n-1} \neg(Fly_i(MI, GE) \wedge Drive_{i+1}(GE, SL))$$

[Kautz, Selman 1996], [Biere, et al. 1999], [Latvala, et al. 2004]

Classical Planning with (hard) Constraints

Example (State (Qualification) Constraints)

It is not possible to be in two locations at the same time. Just add to φ:

$$\wedge_{i=0}^{n} \neg(At_i(GE) \wedge At_i(SL))$$

Example (Trajectory Constraints on Action Variables)

I do want to drive after flying. Just add to φ:

$$\wedge_{i=0}^{n-1} \neg(Fly_i(MI, GE) \wedge Drive_{i+1}(GE, SL))$$

[Kautz, Selman 1996], [Biere, et al. 1999], [Latvala, et al. 2004]

Classical Planning with (hard) Constraints

Example (State (Qualification) Constraints)

It is not possible to be in two locations at the same time. Just add to φ:

$$\wedge_{i=0}^{n} \neg (At_i(GE) \wedge At_i(SL))$$

Example (Trajectory Constraints on Action Variables)

I do want to drive after flying. Just add to φ:

$$\wedge_{i=0}^{n-1} \neg (Fly_i(MI, GE) \wedge Drive_{i+1}(GE, SL))$$

[Kautz, Selman 1996], [Biere, et al. 1999], [Latvala, et al. 2004]

Outline

"Soft" goals

Example

Over than satisfying the goal, I have the additional "soft goal" that I have to be in Genova before time 3.
My preference is:

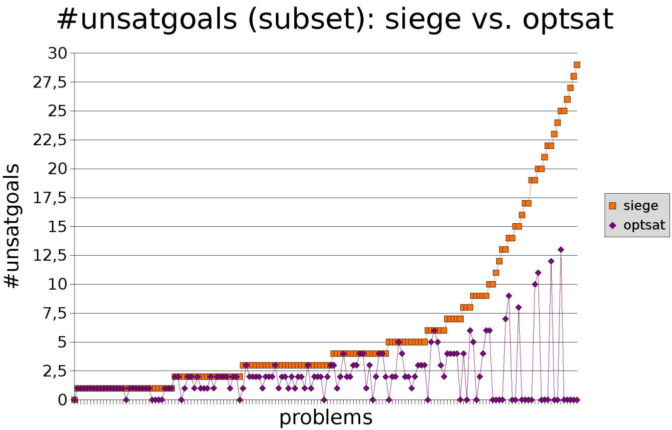
$$At_0(GE) \vee At_1(GE) \vee At_2(GE)$$

Simple preference among goals

1. $\langle I, tr, G\rangle$ is a planning problem
2. S is a set of "soft" goals.
3. Consider $I_0 \wedge \wedge_{i=1}^{n} tr_i \wedge G_n \wedge_{g \in S} (v_n(g) \equiv g_n)$ together with the simple preference $\{v_n(g) : g \in S\}$.

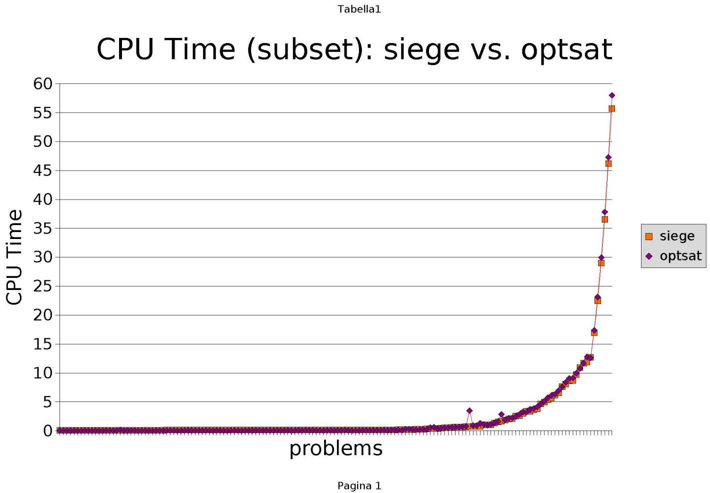
Simple preference among goals: # of unsatisfied goals

Tabella1



Pagina 1

Simple preference among goals: CPU times



Outline

Preferences on the actions

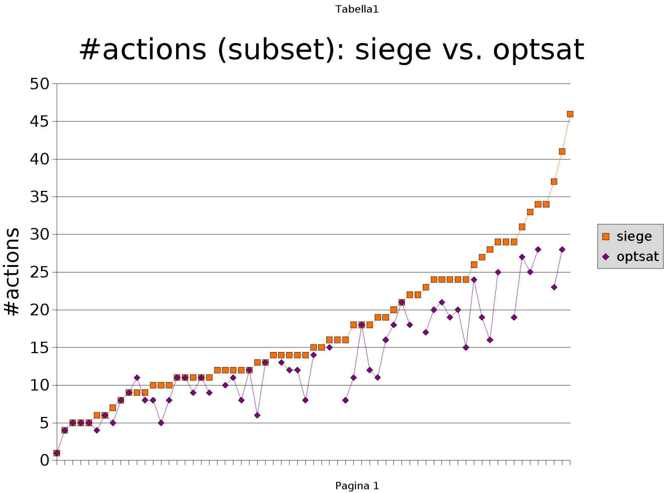
Example

Considering the goal $At_2(MI)$, I want to perform as few actions as possible. Further, it is more important not to take the plain than not taking the car.
My preferences are $\{\neg Drive_1(GE, MI), \neg Plain_1(GE, MI)\}$ with $\neg Plain_1(GE, MI) \prec \neg Drive_1(GE, MI)$.

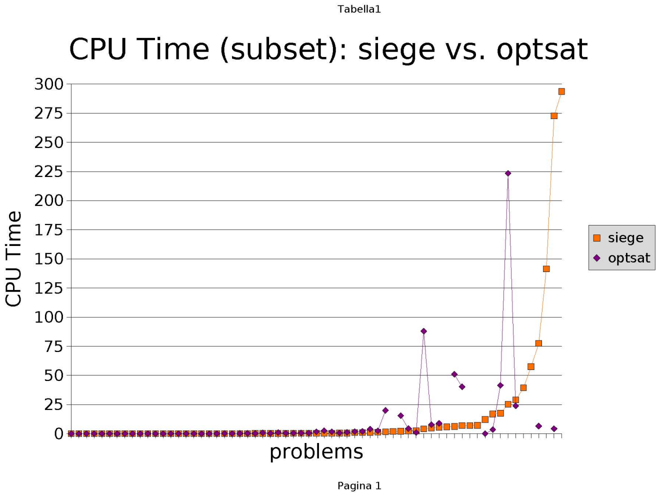
Irredundant plans

1. I want to perform irredundant plans, i.e., plans s.t. there no exists another plan requiring the execution of a subset of the actions.
2. Consider $I_0 \wedge \wedge_{i=1}^{n} tr_i \wedge G_n$ together with the preference $\{\neg a_i : a \text{ is an action}\}$.

Irredundant plans: #actions



Irredundant plans: CPU times



Outline

Outline

1. Propositional Satisfiability (SAT)
2. Preferences
   - Computing one/all optimal solution(s) by guiding search
   - Computing one/all optimal solution(s) by generate&test
   - Experimental results
3. Planning as satisfiability
   - Automated Symbolic Planning
   - Soft Goals
   - Plan quality
4. Automatic Test Generation
   - Automatic Test Generation with CBMC
   - Experimental Analysis

CBMC

CBMC is a bounded model checker that formally verifies ANSI-C programs. The properties checked include

- pointer safety,
- array bounds
- user-provided assertions

The information about the program and the properties to be verified are translated into obtain a Boolean formula that is checked by a SAT procedure.

If the formula is satisfiable (meaning a property has been violated), CBMC generates a counterexample (an error trace) for the property violated

CBMC

CBMC is a bounded model checker that formally verifies ANSI-C programs. The properties checked include

- pointer safety,
- array bounds
- user-provided assertions

The information about the program and the properties to be verified are translated into obtain a Boolean formula that is checked by a SAT procedure.

If the formula is satisfiable (meaning a property has been violated), CBMC generates a counterexample (an error trace) for the property violated

CBMC

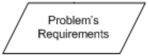
CBMC is a bounded model checker that formally verifies ANSI-C programs. The properties checked include

- pointer safety,
- array bounds
- user-provided assertions

The information about the program and the properties to be verified are translated into obtain a Boolean formula that is checked by a SAT procedure.

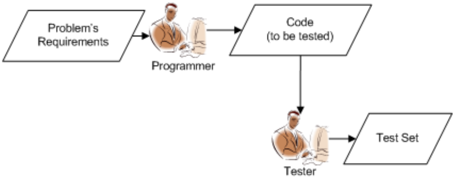
If the formula is satisfiable (meaning a property has been violated), CBMC generates a counterexample (an error trace) for the property violated

Test Generation - Setting

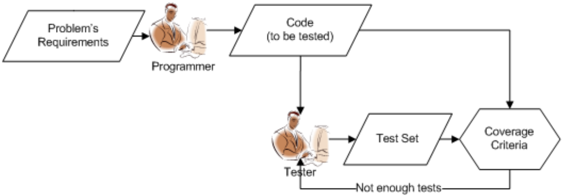


Test Generation - Setting

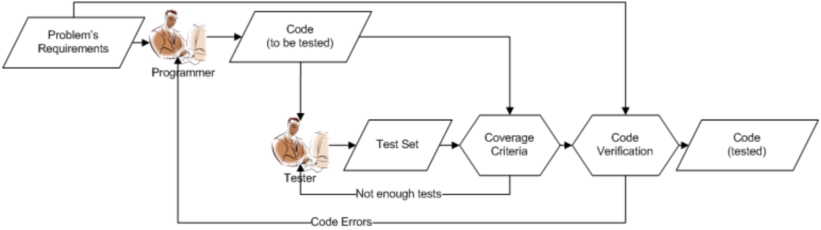
Test Generation - Setting



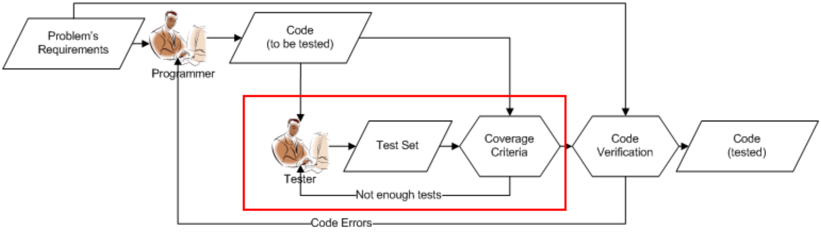
Test Generation - Setting



Test Generation - Setting



Test Generation - Setting



Test Generation - Coverage

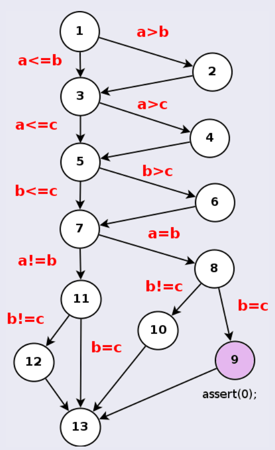
Branch Coverage

Branch coverage requires every possible outcome of every decision of the program to be tested at least once by a test in the set.

Applications

In many safety critical applications, a test set with 100% coverage has to be provided along with the code.

ATG for coverage analysis via user assertion

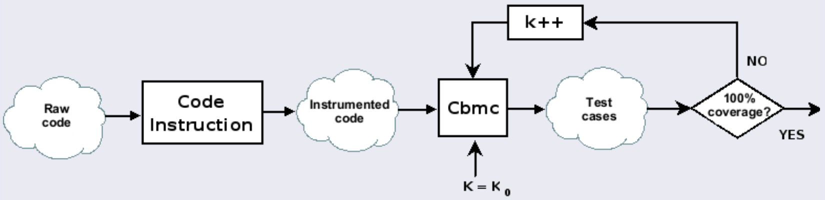


To cover all blocks

- Add an assert(0) statement at the end of a block of instructions.
- Run Cbmc.
- If the assert is reached Cbmc generates an error trace.
- Repeat for each block in the program.

Use the informations from the error trace to generate the test set.

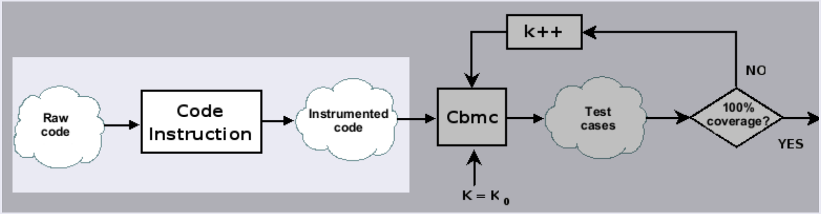
ATG for Coverage Analysis using CBMC



Three main phases

- Code instrumentation
- Test Generation
- Coverage Analysis

Code instrumentation



Code instrumentation

	int FUT(*int a*)
s_0	$int\ r = i = 0$
b_0, b_1	**while** $i < max$ **do**
s_1	$g{+}{+}$
b_2	**if** $i > 0$ **then**
s_2	$a{+}{+}$
b_4, b_5	**if** $a \neq 0$ **then**
s_3	$r = r + \frac{(g+2)}{a}$
b_3	**else**
s_4	$r = r + g + i$
s_5	$i{+}{+}$
s_6	$r = r * 2$
s_7	**return** r

Code instrumentation

int FUT(*int a*)
s_0 $\quad$ $int\ r = i = 0$
b_0, b_1 $\quad$ **while** $i < max$ **do**
s_1 $\quad\quad$ $g{+}{+}$
b_2 $\quad\quad$ **if** $i > 0$ **then**
s_2 $\quad\quad\quad$ $a{+}{+}$
b_4, b_5 $\quad\quad\quad$ **if** $a \neq 0$ **then**
s_3 $\quad\quad\quad\quad$ $r = r + \frac{(g+2)}{a}$
b_3 $\quad\quad$ **else**
s_4 $\quad\quad\quad$ $r = r + g + i$
s_5 $\quad\quad$ $i{+}{+}$
s_6 $\quad$ $r = r * 2$
s_7 $\quad$ **return** r

int FUT(*int a*)
$\quad$ **ASSERT_1**
s_0 $\quad$ $int\ r = i = 0$
b_0, b_1 $\quad$ **while** $i < max$ **do**
$\quad$ **ASSERT_2**
s_1 $\quad\quad$ $g{+}{+}$
b_2 $\quad\quad$ **if** $i > 0$ **then**
$\quad\quad$ **ASSERT_3**
s_2 $\quad\quad\quad$ $a{+}{+}$
b_4 $\quad\quad\quad$ **if** $a \neq 0$ **then**
$\quad\quad\quad$ **ASSERT_4**
s_3 $\quad\quad\quad\quad$ $r = r + \frac{(g+2)}{a}$
b_5 $\quad\quad\quad$ **else**
$\quad\quad\quad$ **ASSERT_5**
b_3 $\quad\quad$ **else**
$\quad\quad$ **ASSERT_6**
s_4 $\quad\quad\quad$ $r = r + g + i$
s_5 $\quad\quad$ $i{+}{+}$
$\quad$ **ASSERT_7**
s_6 $\quad$ $r = r * 2$

Code instrumentation

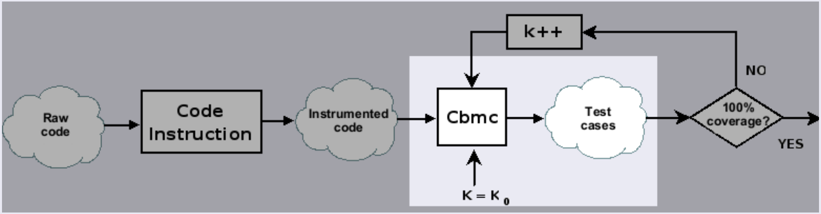
```
#ifdef ASSERT_i
  assert(0)
# endif


int NONDET_INT()

int MAIN(int argc, char * argv[ ])
      max = NONDET_INT()
      g = NONDET_INT()
      int a = NONDET_INT()
      return fut(a)
```

```
         int FUT(int a)
               ASSERT_1
s0             int r = i = 0
b0, b1         while i < max do
               ASSERT_2
s1                g++
b2                if i > 0 then
                  ASSERT_3
s2                   a++
b4                   if a ≠ 0 then
                     ASSERT_4
s3                      r = r + (g+2)/a
b5                   else
                     ASSERT_5
b3                else
                  ASSERT_6
s4                   r = r + g + i
s5                i++
               ASSERT_7
s6             r = r * 2
```

Test Generation



- Choose a k value as number of iterations for the non-deterministic loops
- Run cbmc n times with n=number of assertions added
- Recover from the error trace the informations to generate the test

Outline

1. Propositional Satisfiability (SAT)
2. Preferences
   - Computing one/all optimal solution(s) by guiding search
   - Computing one/all optimal solution(s) by generate&test
   - Experimental results
3. Planning as satisfiability
   - Automated Symbolic Planning
   - Soft Goals
   - Plan quality
4. Automatic Test Generation
   - Automatic Test Generation with CBMC
   - Experimental Analysis

Experimental Analysis

- We have experimented our methodology on 3 modules of the European Rail Traffic Management System (ERTMS) software written by Ansaldo STS.
- 1000 lines of code with several functions for each module.
- Ansaldo STS has estimated that for each test manually generated are necessary 15 minutes.

Experimental Analysis

		Cbmc		Ansaldo STS	
	# function	# test	time(s)	# test	time(s)
mod1	19	148	1212	64	57600
mod2	7	47	1444	26	23400
mod3	13	193	6256	80	72000
Total	**39**	**388**	**8912**	**170**	**153000**

- Both manual and the Automatic Test Generation accomplish the 100% of Branch Coverage
- The Generation time for our methodology is an order of magnitude smaller than the manual generation.
- The number of the test automatically generated is more than double compared to the manual generation

Experimental Analysis

		Cbmc		Ansaldo STS	
	# function	# test	time(s)	# test	time(s)
mod1	19	148	1212	64	57600
mod2	7	47	1444	26	23400
mod3	13	193	6256	80	72000
Total	**39**	**388**	**8912**	**170**	**153000**

- Both manual and the Automatic Test Generation accomplish the 100% of Branch Coverage
- The Generation time for our methodology is an order of magnitude smaller than the manual generation.
- The number of the test automatically generated is more than double compared to the manual generation

Experimental Analysis

		Cbmc		Ansaldo STS	
	# function	# test	time(s)	# test	time(s)
mod1	19	148	1212	64	57600
mod2	7	47	1444	26	23400
mod3	13	193	6256	80	72000
Total	**39**	**388**	**8912**	**170**	**153000**

- Both manual and the Automatic Test Generation accomplish the 100% of Branch Coverage
- The Generation time for our methodology is an order of magnitude smaller than the manual generation.
- The number of the test automatically generated is more than double compared to the manual generation