External plugins in WASP

2016-08-04

1 Answer Set Computation in WASP

WASP computes an answer set by applying the algorithm depicted in Figure 1. In a nutshell, WASP accepts as input a propositional program Π specified in the numerical format of GRINGO and returns as output COHERENT if Π is coherent, INCOHERENT otherwise.

An important feature of the GRINGO format is that each atom of Π is associated to a unique natural number, called the *id* of the atom.

Example 1 *Consider the program* Π *containing only the following rule:*

```
{a;b;c;d}.
```

The output of GRINGO *is the following:*

```
3 4 2 3 4 5 0 0
0
2 a
3 b
4 c
5 d
.
.
.
```

where the first line represents the rule `{a;b;c;d}`*, whereas the remaining lines represent the atoms table, that is an association between the name of the atom and its id.* $\lhd$

The first step of the algorithm implemented by WASP is the parsing of the numerical format of GRINGO. During the parsing, an interpretation I is created and initialized to $\emptyset$ and some basic simplifications of the input program,

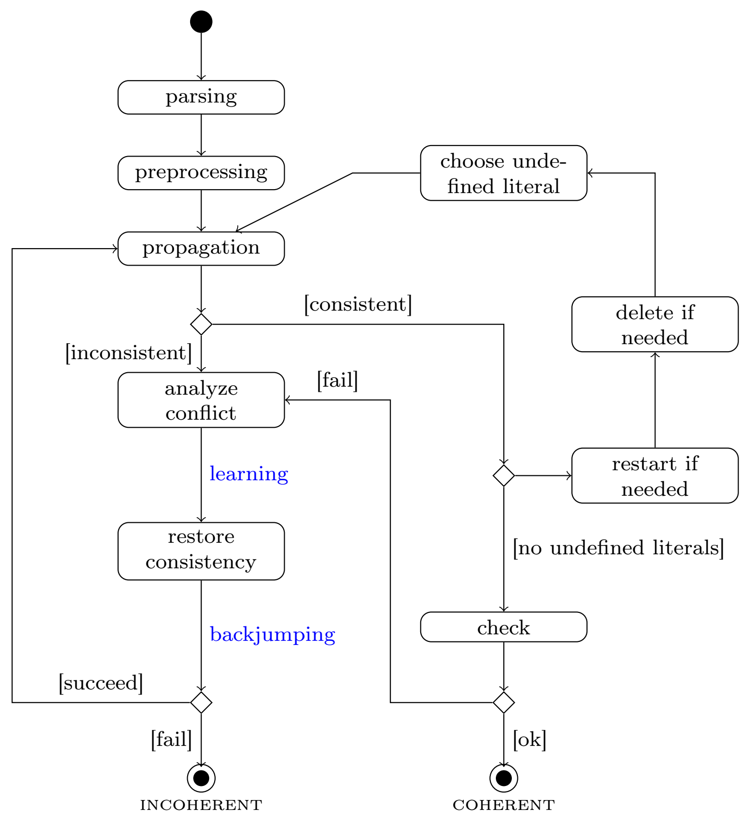

Figure 1: Computation of an answer set in WASP.

such as removing duplicated rules or simplifying facts, are performed. At the end of the parsing, WASP invokes the method `addedVarName` whose role is to notify to the external propagators the correspondence between the name of an atom and its id. Afterwards, WASP invokes the method `getLiterals` which takes as input a set of literals that have been inferred as true during the parsing step (e.g. facts and unary constraints). The output of the method is a list of literals whose notification is required by the propagator. Then, WASP invokes the method `getVariablesToFreeze` that returns as output a list of variables that must be not removed during the subsequent simplification step. In particular, WASP applies polynomial simplifications to strengthen and/or remove redundant rules; or to eliminate atoms by means of clause rewriting. In the latter case WASP invokes the method `onVariableElimination`. During the simplification step, the interpretation I is possibly extended with literals inferred as deterministic consequences. In this case, the corresponding notification is possibly sent to the external propagator (methods `onLiteralTrue` and `onLiteralsTrue`). When all simplifications have been done, WASP invokes the method `simplifyAtLevelZero` whose role is to determine whether some of the literals can be inferred as true by applying specific inferences of the propagator. Afterwards, the backtracking search starts and the method `onStartingSolver` is invoked.

First, I is extended with all the literals that can be deterministically inferred by applying some propagation rules (detailed in Section 1.1). Whenever a literal is inferred as true the corresponding notification is possibly sent to the external propagator (methods `onLiteralTrue` and `onLiteralsTrue`). Three cases are possible after a propagation step is completed: (*i*) I is consistent but not total. In that case, the WASP uses a heuristic strategy to determine whether a restart or a deletion of learned constraints is needed. Then, an undefined literal ℓ (called branching literal) is chosen according to some heuristic criterion, and is added to I. Subsequently, a propagation step is performed that infers the consequences of this choice. (*ii*) I is inconsistent, thus there is a conflict, and I is analyzed. The reason of the conflict is modeled by a fresh constraint r that is added to Π (learning). Moreover, the algorithm backtracks (i.e. choices and their consequences are undone) until the consistency of I is restored (method `onLiteralsUndefined` is invoked). The algorithm then propagates inferences starting from the fresh constraint r. Otherwise, if the consistency of I cannot be restored, the algorithm terminates returning INCOHERENT. Finally, in case (*iii*) I is consistent and total, I is checked (method `checkAnswerSet`). If the check does not retrieve any conflict the algorithm terminates returning COHERENT. Otherwise, the conflict is analyzed and a fresh constraint is possibly added to Π.

1.1 Propagation

The function Propagate extends the interpretation with the literals that can be deterministically inferred. The role of propagation is similar to the unit propagation procedure in the CDCL algorithm, but it is more complex than unit propagation because it implements a set of inference rules for taking in account the properties of ASP programs. In particular, WASP implements three deterministic inference rules for pruning the search space during answer set computation. These propagation rules are named *unit*, *aggregates* and *unfounded*. Unit propagation infers literals appearing in unsatisfied rules containing only one undefined literal. Aggregates propagation concerns the propagation of the truth of aggregate atoms that can be deterministically inferred. Unfounded propagation infers the falsity of atoms appearing in an unfounded set. Moreover, custom propagators can be specified using a PYTHON/PERL interface as detailed in Section 2.

For each literal ℓ in the interpretation unit propagation, aggregates prop-

```
Function Propagate($I$)
1 for $\ell \in I$ do
2     $I$ := UnitPropagation($\ell$);
3     $I$ := AggregatesPropagation($\ell$);
4     $I$ := ExternalPropagation($\ell$);          // method onLiteralTrue
5 $I'$ := ExternalPropagation($I$);               // method onLiteralsTrue
6 if $I' \neq \emptyset$ then $I := I \cup I'$; goto 1;
7 $I'$ := UnfoundedPropagation($I$);
8 if $I' \neq \emptyset$ then $I := I \cup I'$; goto 1;
```

agation and external propagation are applied to extend the deterministic consequences of ℓ (lines 2, 3 and 4 of function Propagate). In particular, the external propagation is done by invoking the method `onLiteralTrue`. When all literals of the interpretation have been propagated the method `onLiteralsTrue` of the external propagators is invoked. If new literals are inferred then unit, aggregates and external propagations are performed again for such literals (line 6). Otherwise, unfounded propagation is applied (line 2). Function Propagate terminates when no new literals can be deterministically inferred.

2 Description of the interface

In the following, we describe the interface for defining custom propagators.

- `def addedVarName(var, name):`
  - **Name:** addedVarName
  - **Description:** this method is invoked while reading the name associated to a variable.
  - **Param var:** the id of the variable.
  - **Param name:** the name associated to var.
  - Optional.
- `def getLiterals(*lits):`
  - **Name:** getLiterals
  - **Description:** this method returns a list of literals whose truth will be notified.

  - **Param lits:** a list of literals whose that have been derived true during the parsing. The first element of the list is the number of variables used in WASP. Literal 1 is used to represent $\perp$ thus it is always false.
  - **Return:** a list of literals attached to the propagator, i.e. when a literal in the list is derived as true a notification is sent to the propagator using either method `onLiteralTrue` or method `onLiteralsTrue`.
  - Required only if one between onLiteralTrue and onLiteralsTrue is used.

- `def getVariablesToFreeze():`
  - **Name:** getVariablesToFreeze
  - **Description:** this method returns a set of variables that must be not removed.
  - Optional.

- `def onVariableElimination(var):`
  - **Name:** onVariableElimination
  - **Description:** this method is invoked when a variable is removed by clause rewriting.
  - **Param var:** the id of the variable that has been removed.
  - Optional.

- `def simplifyAtLevelZero():`
  - **Name:** simplifyAtLevelZero
  - **Description:** this method is invoked before starting the search and returns a list of literals to be inferred at level as deterministic consequence. In order to trigger an incoherence add "1" to the list.
  - **Return:** the literals to infer as deterministic consequences.
  - Optional.

- `def onStartingSolver():`
  - **Name:** onStartingSolver
  - **Description:** this method is invoked when the computation starts (after reading the input and performing the simplifications).

  - Optional.

- `def onLiteralTrue(lit, dl):`
  - **Name:** onLiteralTrue
  - **Description:** this method is invoked whenever a literal becomes true (either by propagation and by choice).
  - **Param lit:** the true literal
  - **Param dl:** the current decision level of the solver
  - **Return:** A list of literals to infer as true. Afterward, the method `getReason` is invoked.
  - Required if checkAnswerSet is not used. Exactly one between `onLiteralTrue` and `onLiteralsTrue` is required.

- `def onLiteralsTrue(*lits):`
  - **Name:** onLiteralsTrue
  - **Description:** this method is invoked after unit propagation and notifies all literals that becomes true (either by propagation and by choice).
  - **Param lits:** the true literals, where the first element is the current decision level of the solver.
  - **Return:** A list of literals to infer as true. Afterward, the method `getReason` is invoked.
  - Required if checkAnswerSet is not used. Exactly one between `onLiteralTrue` and `onLiteralsTrue` is required.

- `def getReason():`
  - **Name:** getReason
  - **Description:** returns a clause modeling the reason for the assignments made by `onLiteralTrue`. That is, let $P = \{p_1, \ldots, p_m\}$ be the literals returned by `onLiteralTrue`, the reason is of the form $\ell_1 \wedge \ldots \wedge \ell_n \rightarrow p$ (for each $p \in P$). The output of this method is a set of literals S of the form $-\ell_1, \ldots, -\ell_n$.
  - **Return:** The set of literals S.
  - Required only if one between onLiteralTrue and onLiteralsTrue is used.

- `def onLiteralsUndefined(*lits):`
  - **Name:** onLiteralsUndefined
  - **Description:** this method is invoked when some of the literals notified as true are again undefined (e.g. after an unroll or a restart).
  - **Param lits:** the undefined literals, where the first element is the current decision level of the solver.
  - Required only if one between onLiteralTrue and onLiteralsTrue is used.
- `def checkAnswerSet(*answerSet):`
  - **Name:** checkAnswerSet
  - **Description:** this method is invoked after an answer set is found. The role of the method is to check whether the answer set is consistent with the propagator.
  - **Param answerSet:** the answer set as list of literals. ℓ if the literal is true in the answer set, $-\ell$ if the literal is false.
  - **Return:** 0 if the answer set is inconsistent, !=0 otherwise.
  - Required only if onLiteralTrue and onLiteralsTrue are not used.
- `def getReasonForCheckFailure():`
  - **Name:** getReasonForCheckFailure
  - **Description:** returns a clause modeling the reason for the failure triggered by `checkAnswerSet`. The output of this method is a set of literals S of the form $\ell_1, \ldots, \ell_n$ interpreted as $\ell_1 \vee \ldots \vee \ell_n$.
  - **Return:** The set of literals S.
  - Required only if checkAnswerSet is used.
- `def storeClauseFromCheckFailure():`
  - **Name:** storeClauseFromCheckFailure
  - **Description:** if this method is used then the clause returned by `getReasonForCheckFailure` is also stored.
  - Required only if checkAnswerSet is used.
- `def onAnswerSet(*answerSet):`

  - **Name:** onAnswerSet
  - **Description:** this method is invoked when a new answer set is found.
  - **Param answerSet:** the answer set as list of literals. ℓ if the literal is true in the answer set, $-\ell$ if the literal is false.
  - Optional.

- `def onNewLowerBound(bound):`
  - **Name:** onNewLowerBound
  - **Description:** this method is invoked only on optimization problems whenever a new lower bound of the optimum solution is found.
  - **Param bound:** the computed lower bound.
  - Optional.

- `def onNewUpperBound(bound):`
  - **Name:** onNewUpperBound
  - **Description:** this method is invoked only on optimization problems whenever a new upper bound of the optimum solution is found.
  - **Param bound:** the computed upper bound.
  - Optional.

3 Usage

In the following we show an example of usage of the interface described in Section 2. In our example we consider again the program of Example 1, i.e. the program containing only the choice rule `{a;b;c;d}.` Here, we want to create a propagator modeling a rule of the type `:- #count{a;b;c;d} != 2`, i.e. exactly two atoms among `a`, `b`, `c`, and `d` must be true. In the following we present two different implementations of the propagator.

The first implementation, listed below, simply counts the number of true (resp. false) literals and infers the remaining ones as false (resp. true) when the number is equal to 2.

```python
interpretation = []
TRUE = 1
FALSE = -1
UNDEFINED = 0
countTrue = 0
countFalse = 0

def getLiterals(*lits):
    global interpretation
    #The first position of lits contains the
    #number of variables used in wasp
    for i in range(0, lits[0]+1):
        interpretation.append(UNDEFINED)
    #Attached to all changes of the truth values
    l = []
    for i in range(2,6):
        l.append(i)
        l.append(-i)
    return l

def getVariablesToFreeze():
    #Freeze all variables
    lits = [2,3,4,5]
    return lits

def onLiteralTrue(lit, dl):
    global interpretation, countTrue, countFalse
    #Count true and false and propagate if needed
    l = abs(lit)
    if lit >= 0:
        countTrue = countTrue + 1
        interpretation[l]=TRUE
    else:
        countFalse = countFalse + 1
        interpretation[l]=FALSE
    output = []
    if countTrue == 2:
        for i in range(2,6):
            if interpretation[i] == UNDEFINED:
                output.append(-i)
    elif countFalse == 2:
        for i in range(2,6):
            if interpretation[i] == UNDEFINED:
                output.append(i)
    return output

def getReason():
    #Let T={a,b} be the true atoms then the reason for
    #the assignment is a ^ b.
    #Thus, a ^ b -> c and a ^ b -> d.
    #The reason is a clause thus literals are flipped.
    reason=[]
    if countTrue == 2:
        for i in range(2,6):
            if interpretation[i] == TRUE:
                reason.append(-i)
    else:
        for i in range(2,6):
            if interpretation[i] == FALSE:
                reason.append(i)
    return reason

def onLiteralsUndefined(*lits):
    global interpretation, countTrue, countFalse
    #If a literal is again undefined
    #the interpretation is restored.
    for i, l in enumerate(lits):
        if(i==0):
            continue
        if(l>=0):
            countTrue = countTrue -1
        else:
            countFalse = countFalse - 1
        interpretation[abs(l)] = UNDEFINED
    return
```

The second implementation, listed below, does not infer any literal during the computation. Instead, it checks that the computed answer set contains exactly two true atoms. If the check fails a reason is computed and added to the solver.

```
reason = []
answer = []
```

```
def checkAnswerSet(*answer_set):
    global answer
    #Check whether the answer set has exactly two atoms true
    count = 0
    for i in range(2,6):
        if answer_set[i] > 0:
            count = count+1
    if count != 2:
        answer = answer_set
        return 0
    return 1

def getReasonForCheckFailure():
    #In case of failure compute the clause
    global answer
    output = []
    for i in range(2,6):
        if answer[i] > 0:
            output.append(-i)
        else:
            output.append(i)
    return output

def storeClauseFromCheckFailure():
    #Store the clauses produced
    pass
```