SAT-Based Answer Set Programming *

Enrico Giunchiglia[1], Yuliya Lierler[2], and Marco Maratea[1]

[1]DIST - Università di Genova, Genova, Italy
[2]Institut für Informatik, Erlangen-Nürnberg-Universität, Germany
[1]{enrico,marco}@mrg.dist.unige.it, [2]yuliya.lierler@informatik.uni-erlangen.de

Abstract

The relation between answer set programming (ASP) and propositional satisfiability (SAT) is at the center of many research papers, partly because of the tremendous performance boost of SAT solvers during last years. Various translations from ASP to SAT are known but the resulting SAT formula either includes many new variables or may have an unpractical size. There are also well known results showing a one-to-one correspondence between the answer sets of a logic program and the models of its completion. Unfortunately, these results only work for specific classes of problems.

In this paper we present a SAT-based decision procedure for answer set programming that (i) deals with any (non disjunctive) logic program, (ii) works on a SAT formula without additional variables, and (iii) is guaranteed to work in polynomial space. Further, our procedure can be extended to compute all the answer sets still working in polynomial space. The experimental results of a prototypical implementation show that the approach can pay off sometimes by orders of magnitude.

Introduction

Propositional satisfiability (SAT) is one of the most studied fields in Artificial Intelligence and Computer Science. Also motivated by the availability of efficient SAT solvers various reductions from logic programs to SAT were introduced in the past.

Fages (1994) showed that if a program Π is "tight" then its answer sets (or stable models) are in one-to-one correspondence with the models of its completion (Clark 1978). If the completion is converted to a set of clauses Γ, state-of-the-art SAT solvers can be used as answer set generators. Since the size of Γ is at most twice the size of Π, and has at most m new variables (where m is the number of rules in the logic program) this is considered a viable and efficient approach. Fages' result was then generalised to include programs with infinitely many rules (Lifschitz 1996), programs tight "on their completion model" (Babovich, Erdem, & Lifschitz 2000), and programs with nested expressions in the bodies of the rules (Erdem & Lifschitz 2003). Still these results do not apply to the whole class of logic programs. It is well known that each answer set corresponds to a model of its completion, but the viceversa in general is not true.

Ben-Eliyahu and Dechter (1996) gave a translation from a class of disjunctive logic programs to SAT. However the translation may need $O(n^2)$ new variables and $O(n^3)$ new clauses (where n is the number of atoms in the logic program). Janhunen (2003) presented an optimized encoding of this translation, which behaves subquadratic in both size and number of atoms.

A reduction to SAT which does not need extra variables was proposed by Lin and Zhao (2002). The drawback of this reduction is that the resulting formula may blow-up in space. Still system ASSAT based on such reduction outperforms state-of-the-art ASP systems like SMODELS (Niemelä 1999; Simons 2000) and DLV (Eiter *et al.* 1998) on many interesting problems.

In this paper the question that we positively answer is: Is it possible to build an efficient SAT-based answer set generator that (i) deals with any (non disjunctive) logic program, (ii) works on a SAT formula without additional variables except for those eventually introduced by the clause form transformation, and (iii) is guaranteed to work in polynomial space? We present a procedure, called ASP-SAT, having the above three but also other features. We integrated ASP-SAT in CMODELS[1] and ran a wide comparative analysis with other state-of-the-art systems. The results show that our procedure has a clear edge over them.

The paper is structured as follows. First we introduce some necessary definitions and terminology. Second we present the main ideas behind our procedure and some details for an effective implementation. We end the paper describing the integration in CMODELS, the experimental results, and the conclusions.

Formal Background

Let P be a set of atoms. A *rule* is an expression of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \qquad (1)$$

*We are grateful to Paolo Ferraris and Vladimir Lifschitz for their comments related to the subject of the paper; to Esra Erdem and Keijo Heljanko for providing us with the benchmarks; and to Francesco Calimeri for his support on DLV. This work is partially supported by ASI, MIUR and Texas Higher Education Coordinating Board under Grant 003658-0322-2001.

[1]http://www.cs.utexas.edu/users/tag/cmodels

where $A_0 \in P \cup \{\bot\}$ ($\bot$ is the logical symbol standing for *False*), and $\{A_1, \ldots, A_n\} \subseteq P$ ($0 \leq m \leq n$). A_0 is the *head* of the rule, $A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$ is the *body*. A *(non disjunctive) logic program* is a finite set of rules.

In order to give the definition of an answer set we consider first the special case in which the program Π does not contain the negation as failure operator not (i.e. for each rule (1) in Π, $n = m$). Let Π be such a program and let X be a set of atoms. We say that X is *closed* under Π if for every rule (1) in Π, $A_0 \in X$ whenever $\{A_1, \ldots, A_m\} \subseteq X$. We say that X is an *answer set* for Π if X is the smallest set closed under Π.

Now consider an arbitrary program Π. Let X be a set of atoms. The *reduct* Π^X of Π relative to X is the set of rules

$$A_0 \leftarrow A_1, \ldots, A_m$$

for all rules (1) in Π such that $X \cap \{A_{m+1}, \ldots, A_n\} = \emptyset$. Thus Π^X is a program without negation as failure. We say that X is an *answer set* for Π if X is an answer set for Π^X.

Our next step is to introduce the relation between the answer sets of Π and the models of its completion. In the following we represent an interpretation in the sense of propositional logic as the set of atoms *True* in it. With this convention a set of atoms X can denote both an answer set and an interpretation.

If A_0 is an atom or the symbol $\bot$, the *completion of* Π *relative to* A_0 is the formula

$$A_0 \equiv \bigvee (A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n)$$

where the disjunction extends over all rules (1) in Π with head A_0. The *completion* *Comp*(Π) of Π consists of the formulas *Comp*(Π, A_0), one for each symbol A_0 in $P \cup \{\bot\}$.

It is well known that if X is an answer set of Π then X satisfies *Comp*(Π) while the converse is not necessarily true. Lin and Zhao (2002) proved that to have a one-to-one correspondence between the answer sets of Π and the models of its completion we have to consider the loop formulas of Π. To state this formally we need the following definitions.

The *dependency graph* of a program Π is the directed graph G such that the vertexes of G are the atoms in Π, and G has an edge from A_0 to $A_1, \ldots, A_m$ for each rule (1) in Π with $A_0 \neq \bot$. A *loop* of Π is a set L of atoms such that for each pair A, A' of atoms in L there is a path from A to A' in the dependency graph of Π whose intermediate nodes belong to L.

Given a loop L, we define $R(L)$ to be the set of formulas

$$A_1 \wedge \ldots \wedge A_m \wedge \neg A_{m+1} \wedge \ldots \wedge \neg A_n$$

for all rules (1) in Π, with $A_0 \in L$ and $\{A_1, \ldots, A_m\} \cap L = \emptyset$. The *loop formula associated with* L is

$$\bigvee L \supset \bigvee R(L)$$

where $\bigvee L$ denotes the disjunction of the elements in L, and similarly for $\bigvee R(L)$. For instance, the only loop formula of the program $\{p \leftarrow p, p \leftarrow \mathit{not}\ q\}$ is $p \supset \neg q$.

Proposition 1 *(Lin & Zhao 2002) Let* Π *be a program, Comp*(Π) *its completion, and* $LF(\Pi)$ *be the set of loop formulas associated with the loops in* Π. *For each set of atoms* X, X *is an answer set of* Π *iff* X *is a model of Comp*(Π) $\cup$ $LF(\Pi)$.

SAT-Based Answer Set Solvers

Consider a program Π. Given Proposition 1 it is clear that if the dependency graph of Π has no cycles (in this case we say that Π is *tight*) then the models of *Comp*(Π) are also answer sets of Π. Thus for tight programs answer set systems can use SAT solvers as "black-box" search engines. CMODELS used this approach to compute answer sets for tight programs.

If Π is non tight, Lin and Zhao (2002) presented the following procedure $LZ(\Pi)$ which still uses SAT solver as black-boxes:

1. Compute *Comp*(Π) and convert it to a set of clauses Γ.
2. Find a model X of Γ by using a SAT solver. Exit with failure if no such model exists.
3. Compute the set of atoms $X^- = X - Cons(\Pi^X)$, where $Cons(\Pi^X)$ is the smallest set of atoms closed under Π^X.
4. If $X^- = \emptyset$, then return X.
5. Otherwise, add the clauses corresponding to the loop formulas of all the maximal (under subset inclusion) loops in X^- to Γ, and go to step 2.

$LZ(\Pi)$ either returns an answer set for Π, or failure if Π does not have answer sets. In their article Lin and Zhao showed that ASSAT, a system implementing the above procedure, can outperform rival systems often by orders of magnitude. Still, $LZ(\Pi)$ has the following two drawbacks:

1. It is not guaranteed to work in polynomial space. In fact, Π can have exponentially many loops: If we assume that each loop formula is not redundant (i.e., that it is not entailed by the rest of the formula under consideration), then
   - If Π has an answer set then $LZ(\Pi)$ blows up in space in the worst case, while
   - If Π has no answer set then $LZ(\Pi)$ is bound to blow up in space: In $LZ(\Pi)$ adding and keeping loop formulas is essential to guarantee that the SAT solver does not return previously computed models, and ultimately to guarantee ASSAT termination.
2. Considering two successive calls of the SAT solver, the computation done for finding the first model is completely discarded. Thus some branches of the search tree may get computed many times.

These drawbacks can be eliminated if we do not use a SAT solver as a black-box. Instead we can take advantage that state-of-the-art complete SAT solvers are based on the Davis-Logemann-Loveland procedure (DLL) (1962). The basic observation is that DLL can easily work as a SAT enumerator. We can thus compute *Comp*(Π) and then

- *generate* models of *Comp*(Π), and
- *test* whether the generated models are answer sets of Π.

$\text{DLL}(\Gamma, S)$
if $\Gamma = \emptyset$ **then return** *True*;
if $\emptyset \in \Gamma$ **then return** *False*;
if $\{l\} \in \Gamma$ **then return** $\text{DLL}(assign(l, \Gamma), S \cup \{l\})$;
$A :=$ an atom occurring in Γ;
return $\text{DLL}(assign(A, \Gamma), S \cup \{A\})$ **or**
$\text{DLL}(assign(\neg A, \Gamma), S \cup \{\neg A\})$.

Figure 1: The DPLL procedure

Consider DLL as in Figure 1, where l denotes a literal; Γ a set of clauses; S an *assignment*, i.e. a consistent set of literals. Given an atom A, $assign(A, \Gamma)$ is the set of clauses obtained from Γ by removing the clauses to which A belongs, and by removing $\neg A$ from the other clauses in Γ. $assign(\neg A, \Gamma)$ is defined similarly. In the initial call to DLL Γ is the set of clauses of which we compute a model and S is the empty set. $\text{DLL}(\Gamma, \emptyset)$ returns *True* whenever Γ is satisfiable, and *False* otherwise.

Given DLL, we can obtain a SAT-based answer set generator for Π by

1. Modifying the first line of DLL in the figure by substituting "**return** *True*" with "**return** *test*(S, Π)", a new function which
   - prints the set *atoms*$(S) = S \cap P$ and returns *True*, if *atoms*(S) is an answer set of Π, and
   - returns *False*, otherwise.
2. Defining a function ASP-SAT(Π), that calls $\text{DLL}(\Gamma, \emptyset)$ where Γ is a set of clauses corresponding to *Comp*(Π). Γ can be computed in many ways. Here, our only assumptions are that (i) Γ signature extends P, and (ii) for each set X of atoms in Γ signature, X satisfies Γ iff $X \cap P$ satisfies *Comp*(Π). Standard conversion methods satisfy such conditions.

Notice that the set S in *test*(S, Π) may be non maximal wrt P, i.e., for some atom A in P, both A and $\neg A$ may not belong to S. Thus, $S \cup \{A\}$ entails *Comp*(Π) and in principle we also need to check if *atoms*$(S \cup \{A\})$ is an answer set of Π. However, this additional check is not needed, as established by the following proposition.

Proposition 2 *Let Π be a program, X, X' be two sets of atoms satisfying Comp(Π). If $X \subset X'$ then X' is not an answer set.*

From the above proposition, and the fact that each answer set is also a model of *Comp*(Π) it follows the correctness and completeness of ASP-SAT(Π).

Proposition 3 *Given a program Π,* ASP-SAT(Π) *returns True if and only if Π has an answer set.*

Moreover ASP-SAT(Π) (i) performs the search on *Comp*(Π) and thus does not introduce any extra variables except for those eventually needed by the clause form transformation; (ii) is guaranteed to work in polynomial space; (iii) can deal with both tight and non tight programs. Further,

- In the case of tight problems each generated model of *Comp*(Π) corresponds to an answer set and thus ASP-SAT(Π) behaves as a standard SAT solver run on *Comp*(Π).
- ASP-SAT(Π) can be easily modified for printing all the answer sets of Π: It is enough to modify *test*(S, Π) in order to return *False* also when *atoms*(S) is an answer set.

Compared to ASSAT, ASP-SAT is guaranteed to work in polynomial space and no computation is ever repeated, also when computing all answer sets. Compared to other answer set solvers like SMODELS and DLV, ASP-SAT has the advantage of being SAT-based and thus it can leverage on the great amount of knowledge available in SAT.

Still, most of the state-of-the-art SAT solvers based on DLL, e.g. MCHAFF (Moskewicz *et al.* 2001), use learning when backtracking. With learning, whenever *False* is returned, a "reason" for the failure has to be computed. Intuitively, a reason is a subset S' of the assignment S such that any assignment extending S' will fail. In order to use SAT solvers with learning, it is thus not enough for *test*(S, Π) to return *False* when S is not an answer set. Indeed, it has also to compute a reason for such failure, i.e., a subset S' of S such that for any maximal assignment S'' (i) extending S' and (ii) entailing *Comp*(Π), *atoms*(S'') is not an answer set of Π. One such set is S itself. However in order to try to maximize the advantages of learning, it is important that S' be as small as possible. Thus, for computing such S', the *test*(S, Π) procedure

1. computes the loop formulas associated with the loops in *atoms*$(S) - Cons(\Pi^{atoms(S)})$,
2. determines a subset of S which falsifies one of the loop formulas computed in the previous step.

In our experiments, with such a simple procedure, we are able to compute reasons which are often less than 1% of the size of S. Of course, the above method for computing reasons, cannot be applied when returning *False* if the goal is to determine all the answer sets and *atoms*(S) is an answer set. In this case, by Proposition 2, the set *atoms*(S) can work as reason.

In the SAT literature, it is well known that learning can produce exponential speed-ups. We now show that ASP-SAT with learning and the method for computing reasons based on loop formulas, may invoke *test*(S, Π) exponentially fewer times than ASP-SAT without learning.

Assume the program Π consists of the two rules[2]

$$A_i \leftarrow A_{i+1} \qquad\qquad A_{i+1} \leftarrow A_i$$

for each $i \in \{0, 2, \ldots, 2k\}$. Then $Comp(\Pi)$ includes $A_i \equiv A_{i+1}$ ($i \in \{0, 2, \ldots, 2k\}$) and we can assume that its clausification Γ consists of the two clauses $(\neg A_i \vee A_{i+1})$,$(A_i \vee \neg A_{i+1})$, for each $i \in \{0, 2, \ldots, 2k\}$. Γ has 2^k models while the only answer set of Π is the empty set:

[2]In this paragraph for simplicity we assume that the clauses corresponding to the reasons returned by *test*(S, Π) are stored and never deleted.

- ASP-SAT without learning or with learning but in which *test*(S,Π) computes *atoms*(S) as reason when S is not an answer set, may generate 2^k assignments entailing $Comp(\Pi)$.
- ASP-SAT with learning and in which *test*(S,Π) computes as reason the subset of S falsifying one of the loop formulas in *atoms*$(S) - Cons(\Pi^{atoms(S)})$, may generate at most k assignments entailing $Comp(\Pi)$.

Still, for such a simple program, the generation and testing of k assignments seems an overkill. Indeed, for programs Π without negation as failure, we know that there exists exactly one answer set, $Cons(\Pi)$. For such programs, ASP-SAT can be easily tuned to directly compute such answer set by first assigning the atoms in P to *False* while branching. It can be proved that with this modification and for programs Π without negation as failure, the first invocation to *test*(S,Π) has $S = Cons(\Pi)$.

Integration in CMODELS

ASP-SAT was implemented on top of the SIMO system (Giunchiglia, Maratea, & Tacchella 2003) and integrated in CMODELS (Lierler & Maratea 2004) by the last two authors. SIMO is a MCHAFF-like SAT solver (Moskewicz *et al.* 2001), and features two-literal watching data structure, 1-UIP learning, and VSIDS heuristics. However, it does not feature the low level optimizations of MCHAFF and thus it is within a factor of 3 slower than MCHAFF. Our implementation of ASP-SAT incorporates all the techniques presented in previous section, including the idea to assign atoms first to *False* while branching.

Still, the integration of ASP-SAT in CMODELS posed some challenges related to CMODELS expressivity. CMODELS uses LPARSE as frontend and thus its input may contain cardinality expressions (also called "constraint literals" in LPARSE manual[3]) and choice rules, two constructs widely used in answer set programming.[4] Operationally CMODELS performs the following steps:

1. Simplifies the given LPARSE program performing preprocessing similar to those involved in SMODELS.
2. Eliminates cardinality expressions by introducing auxilary atoms and rules. Eliminates choice rules in favor of *nested expressions* in the sense of (Lifschitz, Tang, & Turner 1999). This is done using a procedure defined in (Ferraris & Lifschitz 2003).
3. Verifies that the resulting program with nested expressions is tight: the definition of tightness is generalized to such programs in (Erdem & Lifschitz 2003).
4. Forms the program's completion (see (Lloyd & Topor 1984) for the definition of completion of a program with nested expressions) and calls a SAT solver.

For CMODELS the integration implied calling ASP-SAT instead of the SAT solver. As for ASP-SAT we had to take into account that programs with nested expressions do not satisfy Proposition 2. For instance, the program

$$A \leftarrow \textit{not not } A \tag{2}$$

(corresponding to the translation of the choice rule "$\{A\} \leftarrow$") has two answer sets: $\emptyset$, $\{A\}$. The violation of Proposition 2 implied two modifications in our procedure. Consider a program with nested expressions Π. When we are interested in computing all solutions, we have to guarantee that each set S of literals in *test*(S,Π) is maximal. Assuming that the input set of clauses is satisfiable, SIMO always returns maximal assignments but in the signature of the set of clauses resulting after SIMO preprocessing. However SIMO removes tautological clauses in the preprocessing. Tautological clauses can naturally arise during the completion process and removing them may cause the generation of non maximal (wrt the signature of the input program) assignments. By Proposition 2, this is not a problem if Π does not have nested expressions; it may be a problem otherwise. For instance, the completion of the program (2) is $A \equiv \neg\neg A$. $(A \vee \neg A)$ is the tautological clause corresponding to this completion. After the preprocessing, the set of clauses corresponding to the program is empty, and ASP-SAT would not find the answer set $\{A\}$. Therefore, we modified ASP-SAT preprocessing in order to keep tautological clauses. The second modification involved the function *test*(S,Π). It considers loop formulas as defined in (Lee & Lifschitz 2003) for nested programs. In the case *atoms*(S) is an answer set and we are interested in finding all answer sets of Π, *test*(S,Π) returns the entire set S as a reason since any superset or subset of the atoms in S may be an answer set of Π.

Experimental Results

CMODELS2 was comparatively tested against other state-of-the-art systems on a variety of benchmarks. Some of the benchmarks we considered include cardinality constraints and choice rules, and will be called "extended". The systems we considered are SMODELS version 2.27, ASSAT version 1.52 running MCHAFF as SAT solver, DLV release of 2003-05-16. It worths remarking that while SMODELS, ASSAT and CMODELS2 use LPARSE as preprocessor, and thus can be run on the same problems, DLV does not. This explains why DLV appears only in few tables. Further, ASSAT cannot deal with extended programs. Finally, for DLV we have to mention that it is a system specifically designed for disjunctive logic programs, and that very different results can be obtained depending on the specific encoding being used.

All the tests were run on a Pentium IV PC, with 1.8GHz processor, 512MB RAM DDR 266MHz, running Linux. For SMODELS, ASSAT and CMODELS2, the time taken by LPARSE is not counted.[5] Further, each system was stopped after 3600 seconds of CPU time on an instance, or when it exceeded all the available memory: In the tables, these cases

[3] http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz

[4] The input can also contain general weight expressions ("weight literals") However, optimize statements (see LPARSE manual) are not allowed.

[5] Adding the times of LPARSE will not change the picture for DLV when compared to CMODELS2.

		Standard programs			Extended programs	
#b	#s	SMODELS	ASSAT	CMODELS2	SMODELS	CMODELS2
8	i-1	12.32	0.80	1.19	0.81	0.47
11	i-1	71.78	2.97	4.19	2.97	1.01
8	i	40.87	0.89	2.18	1.56	1.40
11	i	71.42	3.17	4.52	3.41	1.16
8	i+1	23.35	0.96	0.97	4.99	0.31
11	i+1	107.48	3.54	3.33	5.21	0.75

Table 1: Blocks world: "#b" is the number of blocks.

are denoted with "TIME" and "MEM" respectively. Otherwise, the tables report the CPU times in seconds needed by each solver to solve the problem, or a "−" to denote an abnormal exit of the program.

We start our analysis considering blocks world planning problems, encoded as both standard and extended logic programs, the latter formulation due to Erdem (2002). The results are represented in Table 1. In the table, (i) the column "#b" represents the number of blocks; (ii) an "i" in the "#s" (standing for "number of steps") column means that the instance corresponds to the problem of finding a plan in "i" steps, where "i" is the minimum integer for which a plan exists. Thus, the instances with "i" and "$i+1$" in the "#s" column admit at least one answer set, while those with "$i-1$" do not have answer sets. These blocks world problems are tight on their completion models (Babovich, Erdem, & Lifschitz 2000), and thus every model of the completion corresponds to an answer set. As it could be expected, SAT-based systems like ASSAT and CMODELS2 perform (sometimes significantly) better than SMODELS, both on standard and extended programs. On standard programs ASSAT performs slightly better than CMODELS2, and this corresponds to the fact that, on average, MCHAFF is better than SIMO.

We also considered Hamiltonian circuit problems on complete graphs, using both the standard encoding of Niemela (1999), and the extended encoding in the "benchmark problems for answer set programming systems"[6]. These problems are particularly interesting because they are non tight and have exponentially many loops. Thus, one would expect these problems to be difficult for ASSAT, but also for CMODELS2 in the case it will generate and then reject (exponentially) many candidate answer sets. The results are in Table 2. As can be observed, on this test set CMODELS2 performs best, being faster (sometimes by orders of magnitude) than all the other solvers both on standard and extended programs.

The problems in Table 3 are real-world non tight problem related to checking requirements in a deterministic automaton, and are described in (Ştefănescu, Esparza, & Muscholl 2003).[7] Two types of problems are considered and encoded in logic programs. The first type is called IDFD and the results on such problems are reported in the first two rows of the table. The second type of problem is called "Morin", and the results are shown on the last three rows. As can be seen,

[6]http://www.cs.engr.uky.edu/ai/benchmark-suite/ham-cyc.sm

[7]These benchmarks are available at http://www.fmi.uni-stuttgart.de/szs/research/projects/synthesis/benchmarks030923.html

	Standard programs				Extended programs	
	SMODELS	ASSAT	DLV	CMOD2	SMODELS	CMOD2
np30c	11.70	1.14	22.08	0.69	0.36	0.36
np40c	62.89	41.81	97.96	1.63	2.48	0.87
np50c	219.56	14.51	314.46	3.37	8.39	1.79
np60c	594.46	48.80	770.07	5.81	20.47	3.41
np70c	1323.61	291.60	1679.12	8.22	39.41	5.87
np80c	2354.28	32.51	3407.35	14.20	75.36	9.18
np90c	TIME	779.06	TIME	22.23	122.53	14.19
np100c	TIME	–	TIME	28.63	185.52	20.76
np120c	TIME	–	TIME	53.33	418.15	41.84

Table 2: Complete graphs. npXc corresponds to a graph with "X" nodes. CMOD2 is CMODELS2

	SMODELS	ASSAT	DLV	CMODELS2
mutex4	33.92	(0)0.62	840.60	(0)0.68
phi4	0.24	(168)2.98	1.44	TIME
mutex2	0.09	(88)1.78		(0)0.12
mutex3	229.57	MEM		(0)24.16
phi3	2.87	(704)236.91		(57)3.91

Table 3: Checking requirements in a deterministic automaton. DLV was not run on the last 3 instances.

CMODELS2 times out on one instance that is easily solved by all the other solvers. This is due to the dimension of the related propositional formula. On the other hand, for any other solver, there are one/two instances on which CMODELS is at least 1 order of magnitude faster. Interestingly, ASSAT blows up in memory on one instance (and also on other instances, on which the other systems time out).

Non tight, extended real-world problems corresponding to the bounded model checking (BMC) of asynchronous concurrent systems (see (Heljanko & Niemelä 2003))[8] are shown in Table 4. As for the blocks world, these problems are about proving a certain property in a given number of steps, represented as the last number in the instance name. The problems in the first five rows do not have answer sets, while the remaining (obtained by incrementing the number of steps) do. Here the results are mixed, and sometimes CMODELS2 performs much worse than SMODELS. On these problems, our standard heuristic is not well suited. Given a program Π, by changing the heuristic in order to

- first assign the atoms occurring within the negation as failure operator, the order and sign of such atoms determined as in SIMO, and
- then assign the remaining atoms first to *False*, the order determined as in SIMO,

we get the better figures represented in the last column, under the label CMODELS'. The idea behind this heuristic is that we should first get to a set of clauses corresponding to a program Π without negation as failure, and then we should try to satisfy the remaining set of clauses by assigning the fewest possible atoms to *True*.

Summing up, the 4 tables show the performances on 45 problems. If for the Table 4 we consider the results in the

[8]http://www.tcs.hut.fi/~kepa/experiments/boundsmodels/

BMC	SMODELS	CMODELS2	CMODELS2’
dp-10.i-02-b11	382.72	1476.72	442.14
dp-10.s-02-b8	15.24	8.20	14.22
dp-12.s-O2-b9	336.03	65.41	137.34
dp-8.i-O2-b9	8.08	12.62	10.69
dp-8.s-O2-b7	1.19	1.02	2.28
dp-10.i-O2-b12	445.47	3295.72	163.15
dp-10.s-O2-b9	28.87	16.07	15.08
dp-12.s-O2-b10	971.50	209.29	46.51
dp-8.i-O2-b10	5.05	40.01	6.44
dp-8.s-O2-b8	1.76	1.99	2.03

Table 4: Bounded Model Checking Problems.

last column, CMODELS2

- times out on 1 problem, while the other systems do not conclude on at least 3 problems;
- performs better than all the three solvers on 30 problems, and on 26 it has at least a factor of 2; and,
- except for the problem on which it times out, CMODELS2 is either the top performer or within a factor of 2 from it.

We also considered the problem of generating all the answer sets. Here the results are less in favor to CMODELS2 when compared to SMODELS, especially on extended programs. We believe this is because of the very naive way in which reasons are computed by *test*(S, Π), especially when *atoms*(S) is an answer set.

Conclusions

We presented a SAT-based procedure that (i) can deal with any logic program (ii) works on a SAT formula without additional variables, (iii) is guaranteed to work in polynomial space. Further, we evidenced that ASP-SAT can be easily modified in order to generate all the answer sets. We showed how to implement ASP-SAT on top of a MCHAFF-like solver, and discussed the modifications needed in the case of extended programs. The experimental evaluation shows that CMODELS2, can have a significant edge over other state-of-the-art systems. Still, we believe that there is a lot of space for improvements, especially in the heuristics, and in the way reasons are computed.

Finally, we believe that ASP-SAT helps in closing the algorithmic gap between answer set and SAT solvers, with beneficial results especially for the former, given the very advanced state of development of the latter.

References

Babovich, Y.; Erdem, E.; and Lifschitz, V. 2000. Fages' theorem and answer set programming. In *Proc. NMR*.

Ben-Eliyahu, R., and Dechter, R. 1996. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence* 12:53–87.

Clark, K. 1978. Negation as failure. In Gallaire, H., and Minker, J., eds., *Logic and Data Bases*. NY: Plenum Press. 293–322.

Ştefănescu, A.; Esparza, J.; and Muscholl, A. 2003. Synthesis of distributed algorithms using asynchronous automata. In *Proc. of CONCUR*, LNCS 2761.

Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *JACM* 5(7).

Eiter, T.; Leone, N.; Mateis, C.; Pfeifer, G.; and Scarcello, F. 1998. The KR system dlv: Progress report, comparisons and benchmarks. In *Proc. KR*.

Erdem, E., and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518.

Erdem, E. 2002. *Theory and applications of answer set programming*. Ph.D. Dissertation, UT at Austin.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science* 1:51–60.

Ferraris, P., and Lifschitz, V. 2003. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*. To appear.

Giunchiglia, E.; Maratea, M.; and Tacchella, A. 2003. (In)Effectiveness of look-ahead techniques in a modern SAT solver. In *Proc. CP*, LNCS 2833.

Heljanko, K., and Niemelä, I. 2003. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming* 3(4&5):519–550.

Lee, J., and Lifschitz, V.. 2003. Loop formulas for disjunctive logic programs. In *Proc. ICLP*.

Lierler, Y., and Maratea, M. 2004. Cmodels-2: SAT-Based Answer Set Solver Enhanced to Non-tight Programs. In *Proc. LPNMR*, 346–350.

Lifschitz, V.; Tang, L. R.; and Turner, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence* 25:369–389.

Lifschitz, V. 1996. Foundations of logic programming. In Brewka, G., ed., *Principles of Knowledge Representation*. CSLI Publications. 69–128.

Lin, F., and Zhao, Y. 2002. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI*.

Lloyd, J., and Topor, R. 1984. Making Prolog more expressive. *Journal of Logic Programming* 3:225–240.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an Efficient SAT Solver. In *Proc. DAC*.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25:241–273.

Janhunen, T. 2003 A counter-based approach to translating normal logic programs into sets of clauses. *Proc. ASP'03 Workshop*, pp. 166–180.

Simons, P. 2000. Extending and implementing the stable model semantics. In *Doctoral dissertation*, 305–316.