Synthesis from Temporal Specifications Using Preferred Answer Set Programming

Stijn Heymans, Davy Van Nieuwenborgh*, and Dirk Vermeir**

Dept. of Computer Science, Vrije Universiteit Brussel,
VUB Pleinlaan 2, B1050 Brussels, Belgium
{sheymans, dvnieuwe, dvermeir}@vub.ac.be

Abstract. We use extended answer set programming (ASP), a logic programming paradigm which allows for the defeat of conflicting rules, to check satisfiability of computation tree logic (CTL) temporal formulas via an intuitive translation. This translation, to the best of our knowledge the first of its kind for CTL, allows CTL reasoning with existing answer set solvers.

Furthermore, we demonstrate how preferred ASP, where rules are ordered according to preference for satisfaction, can be used for synthesizing synchronization skeletons of processes in a concurrent program from a temporal specification. We argue that preferred ASP is put to good use since a preference order can be used to make explicit some of the decisions tableau algorithms make, e.g. declaratively specifying a preference for maximal concurrency makes synthesis more transparent and thus less error-prone.

1 Introduction

Temporal logics [7] are widely used for expressing properties of nonterminating programs. Transformation semantics, such as *Hoare's logic* are not appropriate here since they depend on the program having a final state that can be verified to satisfy certain properties. Temporal logics on the other hand have a notion of (infinite) time and may express properties of a program along a time line, without the need for that program to terminate. E.g., formulas may express that from each state a program should be able to reach its initial state: $\mathsf{AGEF}\mathit{initial}$.

Two well-known temporal logics are *linear temporal logic (LTL)* [7,20] and *computation tree logic (CTL)* [7,9,4], which basically differ in their interpretation of time: the former assumes that time is linear, i.e. for every state of the program there is only one successor state, while time is branching for the latter, i.e. every state may have different successor states, corresponding to nondeterministic choices for the program.

Another knowledge representation framework is *answer set programming (ASP)* [11,3], a logic programming paradigm with a stable model semantics for negation as failure. A *logic program* corresponds to knowledge one wishes to represent, or, more

* Supported by the FWO.

** This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

M. Coppo et al. (Eds.): ICTCS 2005, LNCS 3701, pp. 280–294, 2005.

specifically, to an encoding of a particular problem, e.g. a planning problem [17,6]; the *answer sets* of the program then provide its intentional knowledge, or the solutions of the encoded problem, e.g. a plan for a planning problem.

Under the open answer set, there are some programs that do not have any solutions. There are cases, however, where it is not a feasible strategy to have no answer sets at all, e.g. in large modular programs where different modules are contributed by different parties, there could be only 2 modules that contradict each other, although a majority does not. One would then still like to deduce knowledge that is not related to this contradiction (if one module says a and another one says $\neg a$, but both say b, it is reasonable to keep b as a conclusion, while being unsure about a or $\neg a$ – the normal answer set semantics, however, would yield no answers at all). The *extended answer set semantics* [24] solves this by *defeating* rules with *competing* rules, and thus extracts as much knowledge from the program as possible, while providing alternatives for conflicting rules.

We relate the temporal logic CTL to extended ASP by reducing satisfiability checking of CTL formulas to satisfiability checking of predicates w.r.t. a logic program under the extended answer set semantics. To the best of our knowledge, this is the first account of a translation of CTL reasoning to answer set programming. The translation allows for CTL reasoning through existing answer set solvers.

A related approach, i.e. reasoning with temporal logics through ASP, is taken in [12], where bounded model checking of asynchronous concurrent systems is simulated by computing (normal) answer sets of programs. These results are generalized in [13] where bounded model checking for LTL is translated to ASP. Since LTL and CTL are incomparable, i.e. there are LTL formulas for which no equivalent CTL formula exists, and vice versa, the translation in [13] from LTL to ASP is not applicable to the CTL case that we consider here. Another translation of LTL reasoning to ASP can be found in [22,21] in the context of planning with, among others, temporal constraints or goals.

We take the application of ASP to temporal reasoning a step further by considering ASP as a vehicle for the synthesis of synchronization skeletons of processes in concurrent programs, given a CTL specification. In the literature, synthesis from a temporal logic specification is usually done by tableau-like algorithms, e.g. in [8,1] for a CTL specification or in [18] for a LTL specification, or by a reduction to automata as in [16]. We argue that preferred ASP, i.e. ASP where there is a preference on the satisfaction of rules as defined in [24], can make declaratively explicit some implicit decisions made by those tableau algorithms, resulting in a more transparent synthesis method. More specifically, we discuss how to obtain, using preferred ASP, concurrent programs that are as concurrent as the temporal specification allows.

A preferred ASP approach to synthesis has the further advantage that an implementation is available: in order to illustrate the theoretical results, we use the OLPS solver [19], available for download from `http://tinf2.vub.ac.be/olp`, to synthesize the well-known mutual exclusion problem.

The remainder of the paper is organized as follows. In Section 2, we present the extended and preferred answer set semantics. The simulation of CTL reasoning with extended ASP, as well as its complexity, is discussed in Section 3. Before concluding and giving directions for further research in Section 5, we present in Section 4 a synthesis method from a CTL specification using preferred ASP.

2 Preferred Answer Set Programming

We introduce the extended answer set semantics as in [24]. A *term* is a constant or a variable, where the former will be written lower-case and the latter upper-case. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate name and t_i, $1 \leq i \leq n$, are terms. A *literal* is an atom a or a classically negated atom $\neg a$; an *extended literal* is a literal l or a literal preceded with the *negation as failure* symbol *not*: $not\ l$. A *program* is a finite set of rules $\alpha \leftarrow \beta$ where α is a set of literals with $|\alpha| \leq 1$, i.e. α is empty or a singleton, and β is a finite set of extended literals. We usually denote a rule as $a \leftarrow \beta$ or $\leftarrow \beta$, and we call the latter a *constraint*. The positive part of the body is $\beta^+ = \{l \mid l \in \beta, l \text{ literal}\}$, the negative part is $\beta^- = \{l \mid not\ l \in \beta\}$, e.g. for $\beta = \{a, not\ \neg b, not\ c\}$, we have that $\beta^+ = \{a\}$ and $\beta^- = \{\neg b, c\}$.

For compactness, we assume that the rule $a(\{x_1, \ldots, x_n\}) \leftarrow$ is equivalent with rules $a(x_1) \leftarrow, \ldots, a(x_n) \leftarrow$. We may type arguments as in the rule $p(a : t) \leftarrow$ which stands for $p(a) \leftarrow t(a)$.

A *ground* atom, (extended) literal, rule, or program does not contain variables. Substituting every variable in a program P with every possible constant in P yields the ground program $gr(P)$. All following definitions in this section assume ground programs and ground (extended) literals; to obtain the definitions for unground programs, replace every occurrence of a program P by $gr(P)$, e.g. an extended answer set of an unground P is an extended answer set of $gr(P)$.

The *Herbrand Base* $\mathcal{B}_P$ of a program P is the set of all atoms that can be formed using the language of P. For a set X of literals, we take $\neg X = \{\neg l \mid l \in X\}$ where $\neg\neg a$ is a; X is *consistent* if $X \cap \neg X = \emptyset$. Let $\mathcal{L}_P$ be the set of literals that can be formed with P, i.e. $\mathcal{L}_P = \mathcal{B}_P \cup \neg\mathcal{B}_P$. An *interpretation* I of P is any consistent subset of $\mathcal{L}_P$. For a literal l, we write $I \models l$, if $l \in I$, which extends for extended literals $not\ l$ to $I \models not\ l$ if $I \not\models l$. In general, for a set of extended literals X, $I \models X$ if $I \models x$ for every extended literal $x \in X$. A rule $r : a \leftarrow \beta$ is *satisfied* w.r.t. I, denoted $I \models r$, if $I \models a$ whenever $I \models \beta$, i.e. r is *applied* whenever it is *applicable*. A constraint $\leftarrow \beta$ is satisfied w.r.t. I if $I \not\models \beta$. The set of satisfied rules in P w.r.t. I is the *reduct* P_I.

For a *simple* program P (i.e. a program without *not*), an interpretation I is a *model* of P if I satisfies every rule in P, i.e. $P_I = P$; it is an *answer set* of P if it is a minimal model of P, i.e. there is no model J of P such that $J \subset I$. For programs P containing *not*, the *GL-reduct* w.r.t. an interpretation I is P^I, where P^I contains $\alpha \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P and $\beta^- \cap I = \emptyset$. I is an *answer set* of P if I is an answer set of P^I. A rule $a \leftarrow \beta$ is *defeated* w.r.t. I if there is a *competing* rule $\neg a \leftarrow \gamma$ that is applied w.r.t. I, i.e. $I \models \{\neg a\} \cup \gamma$. An *extended answer set* I of a program P is an answer set of P_I such that all rules in $P \setminus P_I$ are *defeated*. An n-ary predicate p is *satisfiable* w.r.t. a program P iff there is an extended answer set M of P with some $p(x_1, \ldots, x_n) \in M$.

Example 1. The knowledge that one either likes karaoke or not (rules r_1 and r_2), that the karaoke bar is on a boat (r_3), that one is afraid of water (r_4), unless there is a boat, and that a boat is usually, but not necessarily, on the water (r_5), can be represented by the following program:

$r_1 : karaoke \leftarrow not\ \neg karaoke \quad r_2 : \neg karaoke \leftarrow not\ karaoke$
$r_3 : \quad boat \leftarrow karaoke$
$r_4 : \neg water \leftarrow \quad r_5 : \quad water \leftarrow boat$

We have the extended answer sets $M_1 = \{karaoke, boat, water\}$, $M_2 = \{karaoke, boat, \neg water\}$, $M_3 = \{\neg karaoke, \neg water\}$, with reducts $P_{M_1} = P \setminus \{r_4\}$, $P_{M_2} = P \setminus \{r_5\}$, and $P_{M_3} = P$. One sees that in M_1 the rule r_4 is defeated by r_5.

Resolving conflicts by defeating rules leads to different alternative extended answer sets, as in Example 1. Usually however, a user may have some particular preferences on the satisfaction of the rules. As in [24], we impose a strict partial order[1] $<$ on the rules in P, indicating these preferences, which results in an *ordered logic program (OLP)* $\langle P, < \rangle$.This preferential ordering will induce an ordering $\sqsubseteq$ among the possible alternative extended answer sets as follows: for interpretations M and N of P, M is "more preferred" than N, denoted $M \sqsubseteq N$, if $\forall r_2 \in P_N \setminus P_M \cdot \exists r_1 \in P_M \setminus P_N \cdot r_1 < r_2$. Intuitively, for every rule that is satisfied by N and not by M, and which thus appears to be a counterexample for M being better than N, there is a better rule that is satisfied by M and not by N, i.e. M can counter the counterexample of N. We have that M is "strictly better" than N, $M \sqsubset N$, if $M \sqsubseteq N$ and not $N \sqsubseteq M$. An extended answer set is a *preferred answer set* of $\langle P, < \rangle$ if it is minimal w.r.t. $\sqsubset$ among the extended answer sets. An n-ary predicate p is *preferred satisfiable* iff there is a preferred answer set M of P with some $p(x_1, \ldots, x_n) \in M$.

Example 2. Considering Example 1, the knowledge that one is afraid of water may result in the preference relation, $r_4 < r_5$. We have then, using $r_4 < r_5$, that $M_2 \sqsubset M_1$, and $M_3 \sqsubset M_1$ and $M_3 \sqsubset M_2$ since M_3 satisfies all rules, such that M_3 is the preferred answer set.

3 CTL Reasoning with Extended Answer Set Programming

Let AP be the finite set of available proposition symbols. *Computation tree logic (CTL)* formulas are defined as follows:

- every proposition symbol $P \in AP$ is a formula,
- if p and q are formulas, so are $p \wedge q$ and $\neg p$,
- if p and q are formulas, then $\mathsf{EX}p$, $\mathsf{E}(p\ \mathsf{U}\ q)$, $\mathsf{AX}p$, and $\mathsf{A}(p\ \mathsf{U}\ q)$ are formulas.

The semantics of a CTL formula is given by *(temporal) structures*. A structure K is a tuple (S, R, L) with S a countable set of states, $R \subseteq S \times S$ a total relation in S, i.e. $\forall s \in S \cdot \exists t \in S \cdot (s, t) \in R$, and $L : S \rightarrow 2^{AP}$ a function labeling states with propositions. Intuitively, R indicates the permitted transitions between states and L indicates which propositions are true at certain states.

A path π in K is an infinite sequence of states $(s_0, s_1, \ldots)$ such that $(s_{i-1}, s_i) \in R$ for each $i > 0$. For a path $\pi = (s_0, s_1, \ldots)$, we denote the element s_i with π_i. For a structure $K = (S, R, L)$, a state $s \in S$, and a formula p, we inductively define when K is a *model* of p at s, denoted $K, s \models p$:

[1] A strict partial order on X is an anti-reflexive and transitive relation on X.

- $K, s \models P$ iff $P \in L(s)$ for $P \in AP$,
- $K, s \models \neg p$ iff not $K, s \models p$.
- $K, s \models p \wedge q$ iff $K, s \models p$ and $K, s \models q$.
- $K, s \models \mathsf{EX}p$ iff there is a $(s, t) \in R$ and $K, t \models p$,
- $K, s \models \mathsf{AX}p$ iff for all $(s, t) \in R$, $K, t \models p$,
- $K, s \models \mathsf{E}(p\ \mathsf{U}\ q)$ iff there exists a path π in K with $\pi_0 = s$ and $\exists k \geq 0 \cdot (K, \pi_k \models q \wedge \forall j < k \cdot K, \pi_j \models p)$,
- $K, s \models \mathsf{A}(p\ \mathsf{U}\ q)$ iff for all paths π in K with $\pi_0 = s$ we have $\exists k \geq 0 \cdot (K, \pi_k \models q \wedge \forall j < k \cdot K, \pi_j \models p)$.

$K, s \models \mathsf{EX}p$ ($K, s \models \mathsf{AX}p$) can be read as "there is some neXt state where p holds" ("p holds in all next states"), and $K, s \models \mathsf{E}(p\ \mathsf{U}\ q)$ ($K, s \models \mathsf{A}(p\ \mathsf{U}\ q)$) as "there is some path from s along which p holds Until q holds (and q eventually holds)" ("for all paths from s, p holds until q holds (and q eventually holds)").

Some common abbreviations for CTL formulas are $\mathsf{EF}p = \mathsf{E}(true\ \mathsf{U}\ p)$ (there is some path on which p will eventually hold), $\mathsf{AF}p = \mathsf{A}(true\ \mathsf{U}\ p)$ (p will eventually hold on all paths), $\mathsf{EG}p = \neg\mathsf{AF}\neg p$ (there is some path on which p holds globally), and $\mathsf{AG}p = \neg\mathsf{EF}\neg p$ (p holds everywhere on all paths). Furthermore, we have the standard propositional abbreviations $p \vee q = \neg(\neg p \wedge \neg q)$, $p \Rightarrow q = \neg p \vee q$, and $p \Leftrightarrow q = (p \Rightarrow q) \wedge (q \Rightarrow p)$.

A structure $K = (S, R, L)$ *satisfies* a CTL formula p if there is a state $s \in S$ such that $K, s \models p$; we also call K a *model* of p. A CTL formula p is *satisfiable* iff there is a model of p.

Example 3. Consider the expression of *absence of starvation* $t \Rightarrow \mathsf{AF}c$ [4] for a process in a mutual exclusion problem (more about mutual exclusion in Section 4). The formula demands that if a process tries (t) to enter a critical region, it will eventually succeed in doing so (c) for all possible future execution paths.

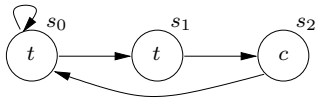


Fig. 1. Example Structure $t \Rightarrow \mathsf{AF}c$

We will usually represent structures by diagrams as in Figure 1, where states are nodes, transitions between nodes define R, and the labels of the nodes contain the propositions true at the corresponding states. The structure K defined by Figure 1 does not satisfy $t \Rightarrow \mathsf{AF}c$ at s_0 since on the path $(s_0, s_0, \ldots)$ c never holds. We have however, $K, s_1 \models t \Rightarrow \mathsf{AF}c$ and $K, s_2 \models t \Rightarrow \mathsf{AF}c$, where the latter holds trivially since $t \notin L(s_2)$.

From a synthesis viewpoint, we are mainly interested in finite structures and thus in n-*satisfiability*, where a CTL formula p is n-*satisfiable* iff there exists a model $K = (S, R, L)$ of p with $|S| = n$, n a non-negative integer. Note that for sufficiently large n, satisfiability is equivalent to n-satisfiability.

Theorem 1 (Small Model Theorem for CTL [7]). *Let p_0 be a CTL formula. Then p_0 is satisfiable iff p_0 has a finite model of size $\leq \exp(\mathrm{length}(p_0))$.*

N-satisfiability of CTL formulas can be reduced to satisfiability of predicates w.r.t. programs under the extended answer set semantics. In order to keep the treatment simple, we will assume that the only allowed temporal constructs are EG, EU, and EX. They are actually adequate in the sense that other temporal constructs can be equivalently, i.e. preserving satisfiability, rewritten using only those three [15]. Before giving the translation of a CTL formula to a program we define the *closure* of a formula, identifying its subformulas. For a formula p, the closure of p is the minimal set $clos(p)$ such that

- $p \in clos(p)$,
- if $\neg q \in clos(p)$, then $q \in clos(p)$,
- if $q \wedge r \in clos(p)$, then $\{q, r\} \subseteq clos(p)$.
- if $\mathsf{EG}q \in clos(p)$, then $q \in clos(p)$,
- if $\mathsf{E}(q\ \mathsf{U}\ r) \in clos(p)$, then $\{q, r\} \subseteq clos(p)$,
- if $\mathsf{EX}q \in clos(p)$, then $q \in clos(p)$.

For a formula p and a non-negative n, we then construct a program consisting of two parts: a generating part G^n and a defining part D^n_p. The program G^n creates n state constants with rule (g_1). The rules (g_2) allow to introduce transitions between states and the rules (g_3) enable any proposition $P \in AP$ to be true at a state or not:

$$state(\{s_0, \ldots, s_{n-1}\}) \leftarrow \qquad (g_1)$$

$$next(S : state, N : state) \leftarrow \qquad \neg next(S : state, N : state) \leftarrow \qquad (g_2)$$

$$[P](S : state) \leftarrow \qquad \neg[P](S : state) \leftarrow \qquad (g_3)$$

where $[P]$ is the predicate corresponding to the proposition P. Finally, in order to make the resulting transition relation total, it imposes the restriction that every state should have a successor: $succ(S) \leftarrow next(S, N)$ and $\leftarrow state(S), not\ succ(S)$ (g_4). The program D^n_p introduces for every non-propositional CTL formula in $clos(p)$ the following rules (we write $[q]$ for the predicate corresponding to the CTL formula $q \in clos(p)$):

$$[\neg q](S) \leftarrow not\ [q](S) \qquad (d_1)$$

$$[q \wedge r](S) \leftarrow [q](S), [r](S) \qquad (d_2)$$

$$[\mathsf{EG}q](S) \leftarrow [q](S), next(S, N), [\mathsf{EG}q]^1(N) \qquad (d^1_3)$$

$$[\mathsf{EG}q]^1(S) \leftarrow [q](S), next(S, N), [\mathsf{EG}q]^2(N) \qquad (d^2_3)$$

$$\vdots$$

$$[\mathsf{EG}q]^{n-1}(S) \leftarrow [q](S), next(S, N), [q](N) \qquad (d^n_3)$$

$$[\mathsf{E}(q\ \mathsf{U}\ r)](S) \leftarrow [r](S) \qquad (d_4)$$

$$[\mathsf{E}(q\ \mathsf{U}\ r)](S) \leftarrow [q](S), next(S, N), [\mathsf{E}(q\ \mathsf{U}\ r)](N) \qquad (d_5)$$

$$[\mathsf{EX}q](S) \leftarrow next(S, N), [q](N) \qquad (d_6)$$

The rules $(d_{\{1,2,6\}})$ are direct translations of the CTL semantics. Rules (d^i_3) ensure there is a finite path of at least $n + 1$ nodes along which q holds; there must be a duplicate s_i on this path which can be used to expand the path into an infinite one.

Rules (d_4) and (d_5) are in accordance with the characterization $\mathsf{E}(q \,\mathsf{U}\, r) \equiv r \vee (q \wedge \mathsf{EXE}(q \,\mathsf{U}\, r))$ [7], and make implicit use of the minimality of answer sets to eventually ensure realization of r.

Combining the two programs, we can reduce n-satisfiability checking for CTL formulas to satisfiability of predicates.

Theorem 2. *Let p be a CTL formula. p is n-satisfiable iff $[p]$ is satisfiable w.r.t. $G^n \cup D^n_p$.*

We call satisfiability checking of a CTL formula using the reduction in Theorem 2, *ASP satisfiability checking*.

Example 4. Consider the formula $t \Rightarrow \mathsf{AF}c$ from Example 3. We have that G^n is the program

$$
\begin{array}{rl@{\qquad}rl}
state(\{s_0, \ldots, s_{n-1}\}) \leftarrow & & & \\
next(S : state, N : state) \leftarrow & & \neg next(S : state, N : state) \leftarrow & \\
[t](S : state) \leftarrow & & \neg[t](S : state) \leftarrow & \\
[c](S : state) \leftarrow & & \neg[c](S : state) \leftarrow & \\
succ(S) \leftarrow & next(S, N) & & \\
\leftarrow & state(S), not\ succ(S) & &
\end{array}
$$

To obtain the defining part of the program, we first rewrite $t \Rightarrow \mathsf{AF}c$ such that it contains only $\neg, \wedge, \mathsf{EG}, \mathsf{EU}$, and EX using the equivalences: $t \Rightarrow \mathsf{AF}c \equiv \neg t \vee \mathsf{AF}c \equiv \neg t \vee \neg\mathsf{EG}\neg c \equiv \neg(t \wedge \mathsf{EG}\neg c)$. The closure of this last formula is $\{\neg(t \wedge \mathsf{EG}\neg c), t \wedge \mathsf{EG}\neg c, t, \mathsf{EG}\neg c, \neg c, c\}$ such that $D^n_{\neg(t \wedge \mathsf{EG}\neg c)}$ is the program

$$
\begin{aligned}
[\neg(t \wedge \mathsf{EG}\neg c)](S) &\leftarrow not\ [t \wedge \mathsf{EG}\neg c](S) \\
[\neg c](S) &\leftarrow not\ [c](S) \\
[t \wedge \mathsf{EG}\neg c](S) &\leftarrow [t](S), [\mathsf{EG}\neg c](S) \\
[\mathsf{EG}\neg c](S) &\leftarrow [\neg c](S), next(S, N), [\mathsf{EG}\neg c]^1(N) \\
[\mathsf{EG}\neg c]^1(S) &\leftarrow [\neg c](S), next(S, N), [\mathsf{EG}\neg c]^2(N) \\
&\vdots \\
[\mathsf{EG}\neg c]^{n-1}(S) &\leftarrow [\neg c](S), next(S, N), [\neg c](N)
\end{aligned}
$$

We then have that the CTL formula $t \Rightarrow \mathsf{AF}c$ is n-satisfiable iff the predicate $[\neg(t \wedge \mathsf{EG}\neg c)]$ is satisfiable w.r.t. $G^n \cup D^n_{\neg(t \wedge \mathsf{EG}\neg c)}$.

Satisfiability checking of CTL formulas is in general EXPTIME-complete [7]. Using the ASP-translation yields a NEXPTIME decision procedure.

Theorem 3. *Let p be a CTL formula. ASP satisfiability checking of p is in* NEXPTIME *w.r.t. the size of p.*

Proof. We can reduce reasoning with extended answer sets to the normal answer set semantics by replacing rules $a \leftarrow \beta$ with $a \leftarrow \beta, not\ \neg a$. Intuitively, if the body is true and the rule cannot be defeated, because the negated head is false, one must apply the rule. Define $E(G^n \cup D^n_p)$ as such a transformed program. From Theorem 4 in

[24] we have that the extended answer sets of $G^n \cup D_p^n$ are exactly the answer sets of $E(G^n \cup D_p^n)$.

Thus $[p]$ is satisfiable w.r.t. $G^n \cup D_p^n$ iff there exists an answer set of $E(G^n \cup D_p^n)$ containing some $[p](s_i)$ iff there exists an answer set of $gr(E(G^n \cup D_p^n))$ containing $[p](s_i)$. By [3], the latter can be done by a nondeterministic Turing Machine in time polynomial in the size of $gr(E(G^n \cup D_p^n))$.

The size of $E(G^n \cup D_p^n)$ is exponential w.r.t. the size of p. Indeed, the number of constants n in $E(G^n \cup D_p^n)$ may be exponential w.r.t. the size of p: by Theorem 1, one may need to introduce an exponential number of states to have equivalence of satisfiability and n-satisfiability. Not considering the rules (g_1) that introduce the constants, and taking $|AP|$ constant, one can see that the size of $E((G^n \cup D_p^n) \backslash g_1)$ is linear in the size of p, as is the size of the closure of p.

Grounding does not yield extra complexity, i.e. the size of $gr(E(G^n \cup D_p^n))$ is polynomial in the size of $E(G^n \cup D_p^n)$[2], resulting in a decision procedure that is in NEXPTIME w.r.t. the size of p. □

Provided EXPTIME $\neq$ NEXPTIME, this result would be less optimal than theoretically attainable for satisfiability checking of CTL formulas. As for Description Logics [2], for which rather efficient solvers, such as FACT [14], exist despite the high theoretical complexity, practical cases can be handled by answer set solvers such as DLV [10], SMODELS [23], or OLPS [19]. Note that the translation in [13] of LTL model checking to ASP is essentially in NEXPTIME as well, since only an exponential bound guarantees that [13]'s bounded model checking coincides with model checking.

Another reasoning problem for CTL is the *Branching-Time Model Checking Problem* [7], which involves checking, given a finite structure $K = (S, R, L)$, whether for each state $s \in S$, $K, s \models p$; if this is the case we call K a *branching-time model* of p[3]. As was the case for satisfiability checking, model checking can also be reduced to computing extended answer sets of a program.

For a structure $K = (S, R, L)$, let M_K be the program

$$state(\{s_0, \ldots, s_{n-1}\}) \leftarrow \qquad \text{for } S = \{s_0, \ldots, s_{n-1}\} \qquad (m_1)$$

$$next(s_i, s_j) \leftarrow \qquad \text{for } (s_i, s_j) \in R \qquad (m_2)$$

$$[P](s_i) \leftarrow \qquad \text{for } P \in L(s_i) \qquad (m_3)$$

i.e. M_K adds the facts defining K.

Theorem 4. *Let $K = (S, R, L)$ be a finite structure and p a CTL formula. K is a branching-time model of p iff $M_K \cup D_p^n \cup \{ \leftarrow not\ [p](s_i) \mid s_i \in S\}$ has an (extended) answer set, for $n = |S|$.*

The component $\{ \leftarrow not\ [p](s_i) \mid s_i \in S\}$ ensures that, for each state $s_i \in S$, $[p](s_i)$ is in every answer set, such that p is satisfied at each state.

[2] In general [5], grounding a program may result in an exponential blow-up, however, $E(G^n \cup D_p^n)$ is such that every rule contains at most 2 different variables and thus contributes to at most n^2 ground rules.

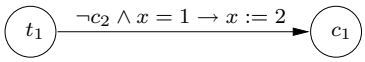
[3] Not to confuse with a model of p, which satisfies only one state.

Satisfiability checking of CTL formulas is EXPTIME-complete, but branching-time model checking for CTL can be done in deterministic polynomial time [7]. Similarly, branching-time model checking for CTL via ASP is one exponential level lower than satisfiability checking via ASP, i.e. in NP.

4 Synthesis from a CTL Specification

We recall some definitions and terminology, see e.g. [8]. A *concurrent program* $P = P_1 \parallel \ldots \parallel P_k$ consists of k processes P_i, $1 \leq i \leq k$, that run in parallel, where the parallelism is, typically, simulated by a nondeterministic interleaving of atomic actions of the processes. We represent processes as *synchronization skeletons*, thereby ignoring any details that are irrelevant to the problem of synchronizing the processes. For example, a process may have a state where it executes some critical code. In synchronization problems, we are then not interested in the actual code that is being executed, but more in the questions whether it is allowed, depending on the state of other processes, to enter the critical section, whether the process is executing the critical section, or whether it is not.

Formally, a synchronization skeleton is a finite state diagram consisting of uniquely labeled states, transitions, and *guarded commands* on the transitions. A guarded command is of the form $B \rightarrow A$, where the *guard* B is a predicate over states or *shared variables* and A is the *command* to be executed. Usually, states are subscripted by the index of the process that is in that particular state, e.g. c_1 indicates that process 1 is in the critical section. For example, the following skeleton



may indicate that process 1 can enter the state c_1 from state t_1 if process 2 is currently not in state c_2 and $x = 1$ for the shared variable x; upon entering the state it executes the command by setting x to 2.

The global computation of the concurrent program can then be seen as a flowgraph system [8], where each state $(s_1, \ldots, s_k, x_1, \ldots, x_m)$ encodes the states s_i its constituting processes are currently in, as well as the value x_j of the m shared variables. For a state $(s_1, \ldots, s_i, \ldots, s_k, x_1, \ldots, x_m)$, a possible next state in the computation of the program is $(s_1, \ldots, s'_i, \ldots, s_k, x'_1, \ldots, x'_m)$ if the i-th process has a transition $s_i \rightarrow s'_i$ labeled by $B \rightarrow A$ such that B is true for $(s_1, \ldots, s_i, \ldots, s_k, x_1, \ldots, x_m)$ and $x'_1, \ldots, x'_m$ represent the values of the shared variables after executing A. Intuitively, a computation step consists of nondeterministically selecting an enabled process (one for which the guard B is true[4]), effectively simulating parallelism. A computation of the program is an infinite path in this flowgraph system.

CTL is used to specify the behavior of the concurrent program, i.e. its flowgraph system, as well as part of the behavior of the processes of the program. *Synthesis* is the task, given a CTL specification, to construct the synchronization skeletons of the processes, and in particular the guarded commands, such that the flowgraph system constructed from these processes satisfies the specification.

[4] We assume, as in [8], nonterminating processes such that there is always an enabled process.

We can distinguish 3 phases in the synthesis method: (1) provide the CTL specification, (2) generate a model if the specification is satisfiable, i.e. the flowgraph system, and (3) define the synchronization skeletons from the flowgraph system. In the sequel, we use (preferred) ASP for the second phase of the synthesis method, for more details on the other phases, we refer the reader to, e.g., [8].

We extend the CTL semantics of Section 3 to better suit the concurrent programming paradigm sketched above. As in [8], we define temporal structures as tuples $K = (S, R_1, \ldots, R_k, L)$ for programs consisting of k processes. The definition of satisfaction for such a structure K is as before with $R = R_1 \cup \ldots \cup R_k$. We introduce the temporal operator $\mathsf{X_i}$, $1 \leq i \leq k$, with $K, s \models \mathsf{EX_i}p$ iff there exists a $(s,t) \in R_i$ such that $K, t \models p$, while $K, s \models \mathsf{AX_i}p$ iff $K, s \models \neg\mathsf{EX_i}\neg p$. Intuitively, $K, s \models \mathsf{EX_i}p$ if there is a transition for process i to a state where p holds. The formula $\mathsf{EX}p$ is equivalent with $\mathsf{EX_1}p \vee \ldots \vee \mathsf{EX_k}p$.

Satisfiability checking of such CTL formulas can be reduced to ASP satisfiability checking of predicates w.r.t. a program $G_k^n \cup D_{p,k}^n$ where G_k^n is the program G^n from Section 3 with (g_2) replaced by k sets of rules (g_2^i), $1 \leq i \leq k$,

$$next_i(S : state, N : state) \leftarrow \qquad \neg next_i(S : state, N : state) \leftarrow \qquad (g_2^i)$$

and rules $next(S, N) \leftarrow next_i(S, N)$ (g_5) added.

The rules (g_2^i) enable the introduction of transitions for individual processes; (g_5) defines the union $R = R_1 \cup \ldots \cup R_k$. The closure of a formula p is modified such that if $\mathsf{EX_i}q \in clos(p)$, then $q \in clos(p)$, and if $\mathsf{EX}q \in clos(p)$, then $\{\mathsf{EX_1}q, \ldots, \mathsf{EX_k}q\} \subseteq clos(p)$. We then obtain the defining part $D_{p,k}^n$ by replacing (d_6) with k rules (d_6^i)

$$[\mathsf{EX_i}\,q](S) \leftarrow next_i(S, N), [q](N) \qquad (d_6^i)$$

Theorem 5. *Let p be a CTL formula. p is n-satisfiable iff $[p]$ is satisfiable w.r.t. $G_k^n \cup D_{p,k}^n$.*

If a CTL formula p is n-satisfiable, we will use the term *flowgraph system* for both a model of p and the corresponding answer set obtained with Theorem 5.

Example 5. Consider 2 synchronization skeletons P_1 and P_2, each modeling the 3 states it can assume: the process can be in the non-critical section of the code (ncs_i, $i \in \{1, 2\}$), it can try to access a critical section of code (try_i), and it can execute the critical section of code (cs_i).

In the *mutual exclusion* problem, one searches for the guarded commands of processes P_1 and P_2 such that they cannot both execute the critical section of the code at the same time. There are many CTL specifications around that model the behavior of the concurrent program executing both processes in parallel, see e.g. [7,4,1,15,18]. We repeat the specification of [8]:

1. Initially, both processes are in their non-critical section: $ncs_1 \wedge ncs_2$.
2. Both processes cannot be in the critical section at the same time (*mutual exclusion*): $\mathsf{AG}\neg(cs_1 \wedge cs_2)$.
3. If a process tries to access its critical section it must always eventually succeed in doing so (*absence of starvation*): $\mathsf{AG}(try_i \Rightarrow \mathsf{AF}cs_i)$.

4. Each process is always in exactly one section: $\mathsf{AG}(ncs_i \vee try_i \vee cs_i)$, $\mathsf{AG}(ncs_i \Rightarrow (\neg try_i \wedge \neg cs_i))$, $\mathsf{AG}(try_i \Rightarrow (\neg ncs_i \wedge \neg cs_i))$, $\mathsf{AG}(cs_i \Rightarrow (\neg ncs_i \wedge \neg try_i))$.
5. If a process is in the non-critical section, it will try to access the critical section in the next step (and it will do nothing else): $\mathsf{AG}(ncs_i \Rightarrow (\mathsf{AX_i} try_i \wedge \mathsf{EX_i} try_i))$. Note that $\mathsf{EX_i} try_i$ is necessary to ensure that there is a next state where try_i holds, $\mathsf{AX_i} try_i$ alone would not be sufficient.
6. If a process is trying to access the critical section, then, if it does a move, it will do so into the critical section: $\mathsf{AG}(try_i \Rightarrow \mathsf{AX_i} cs_i)$.
7. If a process is in the critical section, it will move to the non-critical section (and it will do nothing else): $\mathsf{AG}(cs_i \Rightarrow (\mathsf{AX_i} ncs_i \wedge \mathsf{EX_i} ncs_i))$.
8. If process P_j makes a move, process P_i will do nothing, i.e. they are *asynchronous processes*: $\mathsf{AG}(ncs_i \Rightarrow \mathsf{AX_j} ncs_i)$, $\mathsf{AG}(try_i \Rightarrow \mathsf{AX_j} try_i)$, $\mathsf{AG}(cs_i \Rightarrow \mathsf{AX_j} cs_i)$.
9. Some process can always move, i.e. the program is nonterminating: $\mathsf{AGEX} true$.

Let p be the conjunction of the above CTL formulas, rewritten such that it only contains the temporal operators $\mathsf{EX_i}, \mathsf{EU}$, and EG. The program $G_2^9 \cup D_{p,2}^9$, i.e. with 9 states and 2 processes, has then, among others, the two answer sets, or flowgraph systems, in Figure 2 and Figure 3. We listed only propositions in the states, but s_4 and s_5 are different, since they satisfy different temporal formulas.

Figure 2 is the flowgraph system (call it M_g) usually found in literature as a solution to the mutual exclusion problem, but the structure M_b in Figure 3 also satisfies the mutual exclusion specification. M_b only differs from M_g in the missing of transitions (s_1, s_3)

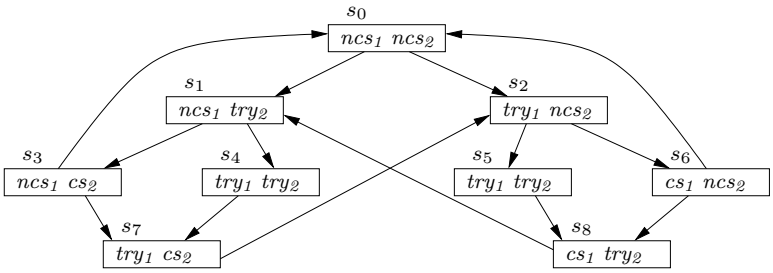


Fig. 2. Maximally Parallel Flowgraph System for Mutual Exclusion

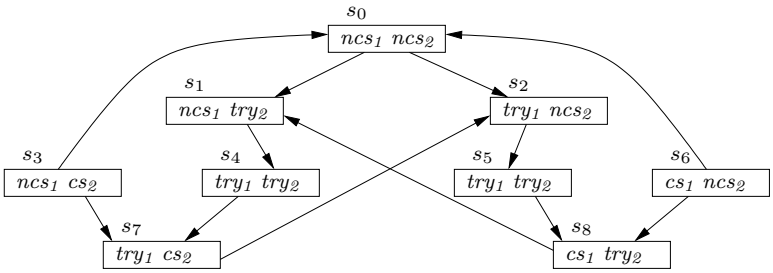


Fig. 3. Flowgraph System for Mutual Exclusion

and (s_2, s_6). It is then natural to wonder why M_g is preferred over M_b as a flowgraph system for the mutual exclusion problem.

The answer is fairly simple. Observing model M_b, one sees that when the system is in the state s_1 its only option is to execute a transition of P_1. This in contrast with M_g where the system can choose between P_1 and P_2. Thus M_g is more nondeterministic, and, by our model of concurrency, allows for more parallelization. That M_b is less parallelized has as a side-effect that P_2 can only enter the critical section if P_1 also tries to enter its critical section, thus, if P_1 decides to do nothing, P_2 is blocked as well. The same scenario cannot occur in M_g since P_2 can go, independently from P_1, from trying to enter to actually entering the critical section.

This drive for more parallelization is implicit in most CTL synthesis methods. E.g., when constructing a model from a tableau, rule [2.2] in [8] says

> *Choose C' to be some $C_j \in Blocks(D_i)$ such that $FRAG[C_i]$ is of minimal size. (Choose one with a maximal number of successors among those C_j with fragments of minimal size, and break ties by choosing the one with lowest index in a predefined ordering.)*

Without going into detail, $FRAG[C_i]$ is a part of the tableau that fulfills eventualities appearing in a node C_i. The relevant part of rule [2.2] for parallelization, as [8] indicates, is choosing nodes of maximal outdegree, since this increases the degree of nondeterministic choice in the model. Instead of leaving this for the model constructing algorithm to take care of, we make this *maximal parallelization* property declaratively explicit.

Definition 1 (Maximal Parallelization Property). *Let p be a CTL formula. A model $M_1 = (S, R_1, \ldots, R_k, L)$ of p is **more parallel** than a model $M_2 = (S, T_1, \ldots, T_k, L)$ of p, denoted $M_1 \preceq M_2$, if $\forall 1 \leq i \leq k \cdot T_i \subseteq R_i$. As usual, we have $M_1 \prec M_2$ if $M_1 \preceq M_2$ and not $M_2 \preceq M_1$.*

*A model $M_1 = (S, R_1, \ldots, R_k, L)$ of p is **maximally parallel** if it is minimal w.r.t. $\prec$. A CTL formula p is **maximally (n-)satisfiable** iff p is (n-)satisfiable by a maximally parallel model.*

It is clear that $\preceq$ is a partial order with $\prec$ its strict version. Intuitively, a model M_1 is more parallel than M_2 if they have the same states with the same labeling of the states, but, for each process, the set of transitions of M_2 is a subset of the set of transitions of M_1.

Example 6. We have that M_b is indeed not maximally parallel since $M_g \prec M_b$. Model M_g on the other hand is maximally parallel. Every state in a model of the CTL specification for mutual exclusion has a maximum of 2 outgoing transitions: process P_i, $i \in \{1, 2\}$, in a given state has only one possibility, i.e. from ncs_i to try_i, from try_i to cs_i, and from cs_i to ncs_i. Thus the only candidates in M_g for the inclusion of more transitions are s_4, s_7, s_5, and s_8.

For s_4 the only possible addition is a transition to s_8. However, this violates the absence of starvation property, since we then have an infinite path $(s_1, s_4, s_8, s_1, \ldots)$ without getting into the critical section for process 2. The transitions going out of s_7 can only be extended by a transition to a state $[cs_1\ cs_2]$, violating the mutual exclusion problem. s_5 and s_8 can be treated similarly.

If a CTL formula is satisfiable, it is always maximally satisfiable.

Theorem 6. *Let p be a CTL formula. p is maximally satisfiable iff p is satisfiable.*

As was the case for normal satisfiability, maximal satisfiability is essentially equivalent to maximal n-satisfiability. The proof is similar to the proof from Theorem 6.

Theorem 7. *Let p be a CTL formula. p is maximally satisfiable iff p is maximally n-satisfiable for an n exponential in the size of p.*

Preferred ASP is well-suited for the expression of such a maximal parallelization property. To obtain maximally parallel models, we define the order $<$ on $G^n_k \cup D^n_{p,k}$ such that $next_i(S : state, N : state) < \neg next_j(S : state, N : state)$ for $1 \leq i, j \leq k$. Intuitively, the program $\langle G^n_k \cup D^n_{p,k}, < \rangle$ attempts to introduce as many transitions as possible with the most preferred $next_i(S : state, N : state) \leftarrow$. It only allows defeat of such preferred rules with the less preferred $\neg next_i(S : state, N : state) \leftarrow$ if the CTL formula would otherwise be unsatisfiable. Theorem 7 ensures that we can restrict ourselves to maximal n-satisfiability.

Theorem 8. *Let p be a CTL formula. p is maximally n-satisfiable iff p is preferred satisfiable w.r.t. $\langle G^n_k \cup D^n_{p,k}, < \rangle$.*

Example 7. For the ordered program $\langle G^9_2 \cup D^9_{p,2}, < \rangle$ with $G^9_2 \cup D^9_{p,2}$ as in Example 5 and $<$ defined as in Theorem 8, we obtain the preferred answer set M_g from Figure 2.

For completeness, we briefly describe how [8] obtains synchronization skeletons from the flowgraph system. One first introduces shared variables for every set of propositions that appears more than once as a label of a state, and then one gives a different value to shared variables that represent different states (with the same label), e.g. the label of s_4 in M_g from Example 5 is updated with $TURN = 1$ and s_5 with $TURN = 2$.

Looking at the flowgraph system in Figure 2, one sees that the state transitions for P_1 are $ncs_1 \rightarrow try_1 \rightarrow cs_1 \rightarrow ncs_1 \rightarrow \ldots$ The guards for those transitions are deduced from the flowgraph system: P_1 goes from try_1 to cs_1 in the flowgraph system for global transitions $[try_1\ ncs_2] \rightarrow [cs_1\ ncs_2]$ or $[try_1\ try_2\ TURN = 1] \rightarrow [cs_1\ try_2]$, resulting in a guard $ncs_2 \vee (try_2 \wedge TURN = 1)$ for the transition $try_1 \rightarrow cs_1$ in the synchronization skeleton for P_1 (with empty command). The complete synchronization skeleton for P_1 is shown in Figure 4.

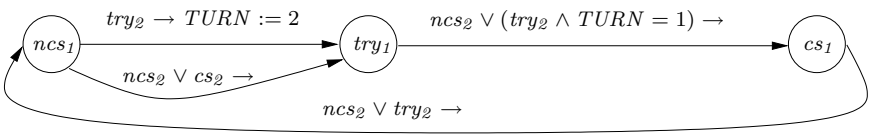


Fig. 4. Synchronization Skeleton for Process P_1

5 Conclusions and Directions for Further Research

We reduced CTL reasoning to extended ASP, investigated the complexity, and indicated where and how preferred ASP can be a useful aid in the synthesis of concurrent programs from a CTL specification.

Noting that both LTL [13] and CTL can be caught within an ASP framework, it is interesting to investigate whether reasoning with the more general temporal logic CTL^* can be reduced to ASP. Another promising application of preferred ASP lies in the debugging of a proposed synthesis for a specification: one can minimally repair the synthesis by defeating faulty state transitions.

References

1. P. C. Attie and E. A. Emerson. Synthesis of Concurrent Programs for an Atomic Read/Write Model of Computation. *ACM Trans. Program. Lang. Syst.*, 23(2):187–242, 2001.
2. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.
5. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
6. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under Incomplete Knowledge. In *Proc. of CL 2000*, volume 1861 of *LNCS*, pages 807–821. Springer, 2000.
7. E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier Science Publishers B.V., 1990.
8. E. A. Emerson and E. M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Sciene of Computer Programming*, 2(3):241–266, 1982.
9. E. A. Emerson and Joseph Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. In *Proc. of the fourteenth annual ACM symposium on Theory of Computing*, pages 169–180. ACM Press, 1982.
10. W. Faber, N. Leone, and G. Pfeifer. Pushing goal derivation in DLP computations. In *Logic Programming and Non-Monotonic Reasoning*, volume 1730 of *LNAI*, pages 177–191. Springer Verslag, 1999.
11. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, Cambridge, Massachusetts, 1988. MIT Press.
12. K. Heljanko and I. Niemelä. Answer Set Programming and Bounded Model Checking. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming*, pages 90–96. AAAI Press, 2001.
13. K. Heljanko and I. Niemelä. Bounded LTL Model Checking with Stable Models. In *Proc. of LPNMR 2001*, volume 2173 of *LNAI*, pages 200–212. Springer, 2001.
14. I. Horrocks. The FaCT system. In *Proc. of Tableaux'98*, volume 1397 of *LNAI*, pages 307–312. Springer, 1998.
15. M. R. A. Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.
16. O. Kupferman and M. Vardi. Synthesis with Incomplete Information. In *Proc. of ICTL 1997*, 1997.

17. V. Lifschitz. Answer Set Programming and Plan Generation. *Journal of Artificial Intelligence*, 138(1-2):39–54, 2002.
18. Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68–93, 1984.
19. D. Van Nieuwenborgh, S. Heymans, and D. Vermeir. An Ordered Logic Program Solver. In *Proc. of PADL 2005*, LNCS. Springer, 2005. To Appear.
20. A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *J. ACM*, 32(3):733–749, 1985.
21. T. C. Son and E. Pontelli. Planning with Preferences Using Logic Programming. In *Proc. of LPNMR 2004*, volume 2923 of *LNCS*, pages 247–260. Springer, 2004.
22. T.C. Son, C. Baral, and S. A. McIlraith. Planning with Different Forms of Domain-Dependent Control Knowledge - An Answer Set Programming Approach. In *Proc. of LPNMR 2001*, volume 2173 of *LNCS*, pages 226–239. Springer, 2001.
23. T. Syrjänen and I. Niemelä. The smodels system. In *Proc. of LPNMR 2001*, volume 2173 of *LNCS*, pages 434–438. Springer, 2001.
24. Davy Van Nieuwenborgh and Dirk Vermeir. Preferred Answer Sets for Ordered Logic Programs. In *Proc. of JELIA 2002*, volume 2424 of *LNAI*, pages 432–443. Springer, 2002.