The Complexity of Model Checking Concurrent Programs against CTLK Specifications

Alessio Lomuscio, Franco Raimondi
Department of Computer Science
University College London – London, UK
{a.lomuscio,f.raimondi}@cs.ucl.ac.uk

ABSTRACT

This paper presents complexity results for model checking concurrent programs against temporal-epistemic formulae. We apply these results to evaluate the complexity of verifying programs by means of two model checkers for multi-agent systems: MCMAS and Verics.

General Terms

Algorithms; Theory; Verification

Keywords

Model checking multi-agent systems; Complexity

1. PRELIMINARIES

1.1 Temporal logics and model checking

The language of Computational Tree Logic (CTL, [7, 4]) is defined over a set of atomic formulae $AP = \{p, q, \dots\}$ as follows:

$$\varphi ::= p | \neg\varphi | \varphi \vee \varphi | EX\varphi | E[\varphi U \psi] | EG\varphi.$$

Other temporal operators to express eventuality and universality can be derived in standard way. We refer to [4] for more details.

CTL formulae are interpreted in *Kripke models*. A Kripke model M for CTL is a tuple $M = (S, R, V, I)$ where S is a set of states, $R \subseteq S \times S$ is a serial transition relation (the *temporal* relation), $V : S \rightarrow 2^{AP}$ is an evaluation function, and $I \subseteq S$ is a set of initial states. *Satisfiability* of a CTL formula φ at a state $s \in S$ in a given CTL model M is defined in a standard way, see [4] for more details.

Model checking is the problem of establishing whether or not a formula φ is satisfied on a given model M. While this check may be defined for a model M of any logic, traditionally the problem of model checking has been investigated mainly for temporal logics. In practical instances, when using model checkers, states and relations in temporal models are not listed *explicitly*. Instead, a *compact description* is usually given for a model M. Various techniques are available to give these specifications, a popular being *concurrent programs* [6]. Concurrent programs offer a suitable framework to investigate the computational complexity of model checking when compact representations are used because, as exemplified in Section 3, various techniques can be reduced as accepting concurrent programs as input.

Formally, a program in this setting is a tuple $D =< AP, AC, S, \Delta, s^0, L >$, where AP is a set of atomic propositions, AC is a set of actions, S is a set of states, $\Delta : S \times AC \rightarrow S$ is a transition function, s^0 is an initial state, and $L : S \rightarrow 2^{AP}$ is a valuation function. Given n programs $D_i =< AP_i, AC_i, S_i, \Delta_i, s_i^0, L_i > (i \in \{1, \dots, n\})$, a concurrent program $D_C = < AP_C, AC_C, S_C, \Delta_C, s_C^0, L_C >$ is defined as the parallel composition of the n programs D_i, as follows: $AP_C = \cup_{1 \le i \le n} AP_i$; $AC_C = \cup_{1 \le i \le n} AC_i$; $S_C = \prod_{1 \le i \le n} S_i$; $(s, a, s') \in \Delta_C$ iff (i) $\forall 1 \le i \le n$, if $a \in AC_i$, then $(s[i], a, s'[i]) \in \Delta_i$, where $s[i]$ is the i-th component of a state $s \in S$; (ii) if $a \notin AC_i$, then $s[i] = s'[i]$; $L_C(s) = \cup_i L_i(s[i])$ (in the remainder, we will drop the subscript C when this is clear from the context). CTL formulae can be interpreted on a (concurrent) program D by using the standard Kripke semantics for CTL formulae. By slight abuse of notation, we will sometimes refer to the programs D_i and to D with the term "Kripke models".

Traditionally, the complexity of temporal logics model checking has been investigated assuming that models are given explicitly. Following this approach, the complexity is given as a function of the size of the model and of the formula being checked. In the case of CTL, the problem of model checking is P-complete [3]. Instead, the complexity of model checking concurrent programs against CTL specifications is investigated in [6] where it is shown that model checking is a PSPACE-complete problem.

1.2 CTLK

CTLK is an extension of CTL with epistemic operators $K_i, i \in \{i, \dots, n\}$ [5]. The formula $K_i\varphi$ expresses the fact that agent i *knows* φ. **CTLK** formulae may be interpreted in a Kripke model $M = (W, R_t, \sim_1, \dots, \sim_n, V)$ where W is a set of states, $R_t \subseteq S \times S$ is a serial transition relation (the *temporal* relation), $\sim_i \subseteq S \times S$ are equivalence relations (the *epistemic* relations), and $V : S \rightarrow 2^{AP}$ is an evaluation function for a given set AP of atomic propositions. Formulae are interpreted in a standard way, by extending the interpretation of CTL formulae with the following:

$M, w \models K_i\varphi$ iff for all $w' \in W, w \sim_i w'$ implies $M, w' \models \varphi$,

CTLK formulae can be interpreted in concurrent programs as well: the temporal operators of **CTLK** are interpreted as in [6], while epistemic operators are evaluated by defining epistemic accessibility relations based on the equality of the components of the

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.

states of a concurrent program (a similar approach can be found in [5]).

2. COMPLEXITY OF MODEL CHECKING CONCURRENT PROGRAMS

In this section we present a proof for the PSPACE-completeness of the problem of model checking concurrent programs against $\mathbf{CTLK}$ specifications. The following two lemmas will be used:

LEMMA 1. *Given a Kripke model $M = (S, R, V, I)$ for CTL, a state $s \in S$, and a formula φ, $M, s \models EG\varphi$ iff there exists a sequence of states π starting from s of length $|\pi| \geq |M|$ s.t. $M, \pi_i \models \varphi$ for all $0 \leq i \leq |M|$.*

LEMMA 2. *Given a Kripke model $M = (S, R, V, I)$ for CTL, a state $s \in S$, and two formulae φ and ψ, $M, s \models E[\varphi U \psi]$ iff there exists a sequence of states π starting from s s.t. $M, \pi_i \models \psi$ for some $i \leq |M|$, and $M, \pi_j \models \varphi$ for all $0 \leq j < i$.*

THEOREM 1. *Model checking concurrent programs against $\mathbf{CTLK}$ specifications is a PSPACE-complete problem.*

PROOF. Given a $\mathbf{CTLK}$ formula and n programs D_i defining a concurrent program D , we define a deterministic, polynomially-space bounded Turing machine T that halts in an accepting state iff $\neg\varphi$ is satisfiable in D (i.e., iff there exists a state $s \in S$ such that $D, s \models \neg\varphi$). Based on this, we conclude that the problem of model checking is in co-PSPACE. As deterministic complexity classes are closed under complement, we conclude that the problem is PSPACE-complete (the lower bound being given by the complexity of model checking CTL in concurrent programs). T is a multi-string Turing machine whose inputs are the n programs D_i and the formula φ. T operates "inductively" on the structure of the formula φ (see also [2] for similar approaches), by calling other machines ("sub-machines") dealing with a particular logical operator only. The input of T includes the states of the program S_i ($1 \leq i \leq n$), the transition relations, the evaluation functions and all the other input parameters of each Δ_i. This information can be stored on a single input tape, separated by appropriate delimiters, together with the formula φ. T returns "yes" iff there exists a state $s \in S$ such that $D, s \models \neg\varphi$. The machine T iterates over the set of states $s = (s_1, \ldots, s_n)$ and checks whether or not $\neg\varphi$ holds in one of these. If a state is found such that $D, s \models \neg\varphi$, then the machine halts in a "yes" state; if the machine loops over all the states without finding a state satisfying $\neg\varphi$, then T halts in a "no" state.

The REACHABILITY algorithm [9] can be used here to check reachability from initial states; notice that only a polynomial amount of space is needed to store states, as they are the product of states of D_i.

A "main" procedure SATISFIABLE can be defined, which operates recursively on the structure of the formula by calling one of the machines described below. Each machine accepts a state s and a formula, and returns either "NO" (the formula is false at s) or "YES" (the formula is true at s). Notice that each machine can call any of the other machines. The following is a description of the formula-specific machines that may be called by SATISFIABLE:

- The machine T_p for atomic formulae simply checks whether or not a `state` is in L(`state`), where the evaluation L is obtained from the evaluations for each program in the input string; if the proposition is true at `state`, then T_p returns YES, otherwise it returns NO.
- The machine $T_\neg$ for formulae of the form $\psi = \neg\varphi$ calls the appropriate machine for φ and returns the opposite value.
- The machine $T_\vee$ for the disjunction $\psi = \psi' \vee \psi''$ first calls the machine for ψ'. If the result is yes, it outputs yes, otherwise it outputs the result of the machine for ψ''.
- The machine T_{EX} accepts a formula φ and a state as input; the machine iterates over the set of states and for each state it checks whether the state is reachable from the input state; if this is the case, then T_{EX} checks whether or not φ is satisfied. If T_{EX} finds such a state, then it halts in a YES state; otherwise, if no reachable state satisfying φ can be found T_{EX} terminates in a NO state. Notice that this machine uses a polynomial amount of space: the space required to store the value of the state.
- The machine T_{EU} for formulae of the form $\psi = E[\varphi' U \psi'']$ is as follows:

```
T_EU(φ,ψ,state) {
  state2 = (1,1,...,1);
  repeat
    if SATISFIABLE(ψ,state2) then
      if ( state == state2) return YES;
      else
        if ( PATH(state,state2,φ,N) ) then
          return YES;
        else
          move to next state2;
        end if;
      end if;
    else
      move to next state2;
    end if
  until last state2;
  return NO;
}
```

  The machine T_{EU} accepts two formulae and a state. The machine checks whether ψ holds in `state2` and whether `state` and `state2` are the same state. If this is the case, then the machine halts in a YES state. Otherwise, the machine checks whether or not there is a sequence of states from `state` to `state2` such that φ holds along the sequence. This check is performed by the procedure PATH(`state`,`state2`,φ,N), which returns YES if there is such a path, of length at most 2^N. By Lemma 2, we take N to be the logarithm of the size of the model. A recursive algorithm to solve PATH is presented in [9]; this algorithm employs at most space proportional to N, and it can be extended by adding a simple check for the satisfiability of φ. As there can be at most $|\varphi|$ checks, PATH uses at most $O(n \cdot |\{D_i\}_{i \in \{1,\ldots,n\}}| \cdot |\varphi|)$ space (i.e. it operates in PSPACE).
- Based on Lemma 1, a non-deterministic machine NT_{EG} can be defined to guess a sequence of states of length greater than $|\{D_i\}_{i \in \{1,\ldots,n\}}|$ in which φ holds. When (and if) such a sequence is found, the machine returns yes (notice that this machine uses a polynomial amount of space and always halts). As NPSPACE=PSPACE [9], it is possible to build a deterministic machine T_{EG} in PSPACE that returns yes iff there exists a sequence of states of length greater than $|\{D_i\}_{i \in \{1,\ldots,n\}}|$ in which φ holds.
- The machine T_K accepts a formula φ, an index i, and and a state as input; this machine operates similarly to T_{EX}, but uses epistemic relations instead of temporal relations.

Each of the machines above uses at most a polynomial amount of space, and there are at most $|\varphi|$ calls to these machines in each run of T. Thus, T uses a polynomial amount of space. $\square$

This proof differs from the proof of PSPACE-completeness for model checking concurrent programs against CTL specifications presented in [6]. The authors of [6] investigate the complexity of various automata and apply their results to the verification of branching time logics. This technique cannot be easily extended to the verification of temporal *and* epistemic modalities. Thus, the proof above provides an alternative proof of the upper bounds for model checking CTL in concurrent programs, which can be easily extended to CTLK.

3. APPLICATIONS

3.1 The complexity of model checking MCMAS programs

MCMAS [11] is a symbolic model checker for interpreted systems. Interpreted systems [5] provide a semantics for temporal and epistemic operators, based on a system of *agents*. Each agent is characterised by a set of local states, by a set of actions, by a protocol specifying the actions allowed in each local state, and by an evolution function for the local states. MCMAS accepts as input a description of an interpreted system and builds a *symbolic* representation of the model by using Ordered Binary Decision Diagrams. We refer to [5, 10, 11] for more details. An interpreted systems described in MCMAS can be reduced to a concurrent program: each agent can be associated with a program $D_i =< AP_i, AC_i, S_i, \Delta_i, s_i^0, L_i >$, where AC_i is the set of actions for agent i, S_i is the set of local states for agent i, and the evolution function Δ_i is the one provided for the agent.

Conversely, the problem of model checking a formula φ in the parallel composition of n programs $D_i =< AP_i, AC_i, S_i, \Delta_i, s_i^0, L_i >$ can be reduced to an MCMAS program. Indeed, it suffices to introduce an agent for each program, whose local states are S_i and whose actions are AC_i. The transition conditions for the agent can be taken to be Δ_i, augmented with the condition that a transition between two local states is enabled if all the agents including the same action in AC_i perform the transition labelled with the particular action.

Therefore, we conclude that the problem of model checking MCMAS programs is a P-complete problem.

It is worth noticing that the actual implementation of MCMAS requires, in the worst case, an exponential space to perform verification. Indeed, MCMAS uses OBDDs, and it is known [1] that for certain problems OBDDs may have a size which is exponential in the number of variables used, irrespective of the ordering of variables chosen.

3.2 The complexity of model checking Verics programs

Verics [8] is a tool for the verification of various types of timed automata and for the verification of **CTLK** properties in multi-agent systems. In this section we consider only the complexity of verification of **CTLK** properties in Verics using un-timed automata.

A multi-agent system is described in Verics by means of a network of (un-timed) automata: each agent is represented as an automaton, whose states correspond to local states of the agent. In this formalism a single set of action is present, and automata synchronise over common actions.

The reduction from Verics code to concurrent programs is straightforward: each automaton is a program D_i and no changes are required for the parallel composition, and similarly a concurrent program can be seen as a network of automata. Thus, we conclude that the problem of model checking un-timed Verics programs is PSPACE-complete.

Notice that the actual implementation of Verics performs verification by reducing the problem to a satisfiability problem for propositional formulae. Similarly to MCMAS, this reduction may lead to exponential time and space requirements in the worst case.

Acknowledgements

We gratefully acknowledge Mike Wooldridge for the valuable comments on an earlier draft of this paper.

Note

A preliminary version of this paper appears in the proceedings of the workshop *Concurrency Specification and Programming* (CS&P 2005), Ruciane Nida, Poland.

4. REFERENCES

[1] R. E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

[2] Allan Cheng. Complexity results for model checking. Technical Report RS-95-18, BRICS - Basic Research in Computer Science, Department of Computer Science, University of Aarhus, February 1995.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications: A practical approach. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[5] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995.

[6] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.

[7] K. McMillan. *Symbolic model checking: An approach to the state explosion problem*. Kluwer Academic Publishers, 1993.

[8] W. Nabialek, A. Niewiadomski, W. Penczek, A. Pólrola, and M. Szreter. VerICS 2004: A model checker for real time and multi-agent systems. In *Proceedings of the International Workshop on Concurrency, Specification and Programming (CS&P'04)*, volume 170 of *Informatik-Berichte*, pages 88–99. Humboldt University, 2004.

[9] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[10] F. Raimondi and A. Lomuscio. MCMAS - A tool for verification of multi-agent systems. http://www.cs.ucl.ac.uk/staff/f.raimondi/MCMAS/.

[11] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via OBDDs. *Journal of Applied Logic*, 2005. To appear in Special issue on Logic-based agent verification.