Journal of Computer and System Sciences 80 (2014) 375–389

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Branching-time logics with path relativisation

Markus Latte [a,1], Martin Lange [b,*,2]

[a] *Department of Computer Science, Ludwig-Maximilians-University Munich, Germany*
[b] *School of Electrical Engineering and Computer Science, University of Kassel, Germany*

A R T I C L E   I N F O

Article history:
Received 2 November 2010
Received in revised form 2 February 2012
Accepted 4 April 2013
Available online 17 May 2013

Keywords:
Temporal logic
CTL*
Formal languages
Expressive power
Model checking
Satisfiability

A B S T R A C T

We define extensions of the full branching-time temporal logic CTL* in which the path quantifiers are relativised by formal languages of infinite words, and consider its natural fragments obtained by extending the logics CTL and CTL$^+$ in the same way. This yields a small and two-dimensional hierarchy of temporal logics parametrised by the class of languages used for the path restriction on one hand, and the use of temporal operators on the other. We motivate the study of such logics through two application scenarios: in abstraction and refinement they offer more precise means for the exclusion of spurious traces; and they may be useful in software synthesis where decidable logics without the finite model property are required. We study the relative expressive power of these logics as well as the complexities of their satisfiability and model-checking problems.

1. Introduction

Branching-time temporal logics are some of the most well-known and used specification languages for reactive systems. The most prominent ones are the logics CTL [12,15] and CTL* [16]. While CTL has nice algorithmic properties – model checking is P-complete and satisfiability checking is EXPTIME-complete – its expressive power is very weak. CTL* amends this by unifying CTL with the linear-time temporal logic LTL [34]. This way, it can express important properties like fairness which is not possible in CTL. This comes at a certain price, though. Model checking CTL* is naturally at least as expensive as it is for LTL. In fact it is no more expensive either, thus, it is PSPACE-complete [35]. The additional complexity introduced by merging branching-time with linear-time formalisms shows through in satisfiability checking which is 2-EXPTIME-complete [39,17].

Temporal logics in general, and with it such branching-time logics, have become prominent because of the success that model checking – an automatic program verification method for correctness properties specified in temporal logics – has had over the past decades [22]. In particular, model checking was very successful in hardware verification because hardware can be modelled by finite-state systems. Verifying infinite-state systems has become more and more important in the domain of program verification since, and this is mainly due to the growing importance of software in reactive systems. Note that software usually leads to infinite-state systems because of the use of unbounded data structures, recursive functions, etc.

The model-checking complexities mentioned above hold with respect to finite models. Clearly, model-checking infinite-state programs is undecidable in general but it remains decidable for certain classes of infinite-state programs, e.g. pushdown processes, and weak temporal logics like CTL and LTL. It is still just PSPACE-complete for LTL but EXPTIME-complete for CTL [9,41].

* Corresponding author.
E-mail addresses: markus.latte@ifi.lmu.de (M. Latte), martin.lange@uni-kassel.de (M. Lange).

[1] Supported by the Doctoral Program "Program and Model Analysis" of the German Research Foundation (GRK 1480).
[2] The European Research Council has provided financial support under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement No. 259267.

0022-0000/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jcss.2013.05.005

Several extensions and variations of such branching-time logics have been considered since, usually with special purposes in mind: Timed CTL has been introduced in order to verify properties in which real-time effects play a crucial role [2]; action-based CTL considers models with more than one accessibility relation [33]; Graded CTL adds some possibility of counting [7]; etc.

Here we consider extensions of branching-time logics in which the path quantifiers can be relativised to traces that belong to a formal language of ω-words. This defines a hierarchy of extensions parametrised by the class of formal languages which can be used for the relativisation. In Section 2 we introduce these logics – based on CTL, CTL* and the lesser known CTL$^+$ which is known to be only as expressive as CTL [15] but exponentially more succinct [42,1,26].

Section 3 motivates the use of these logics through two scenarios: abstraction and software synthesis. Still, the main focus of this paper are the logics themselves. Here we study the hierarchy of expressive power we obtain from these logics (Section 5) with respect to different classes of formal languages, namely the ω-regular ones, the ω-context-free ones and the ω-visibly pushdown ones. We study their the computational complexity and decidability of their satisfiability problems (Section 6) and of their model-checking problems (Section 7).

2. CTL* with path relativisation

2.1. Transition systems

Models of CTL* with path relativisation are transition systems which – as opposed to ordinary CTL* models and like models of action-based CTL – also have labelled edges and need not be total. Let Σ be a finite alphabet and $\mathcal{P}$ be a countably infinite set of atomic propositions. A *transition system* is a tuple $\mathcal{T} = (\mathcal{S}, \to, \lambda)$ where $\mathcal{S}$ is a set of *states*, $\to \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ is the *transition relation*, and $\lambda : \mathcal{S} \to 2^{\mathcal{P}}$ labels each state with a finite set of propositions that are true in this state. We write $s \xrightarrow{a} t$ instead of $(s, a, t) \in \to$.

In order to simplify technical details, we assume that Σ always contains a special character d and that each transition system has a distinct state end with $s \xrightarrow{\mathsf{d}} \mathsf{end}$ for every s including end itself. Furthermore, end has no other incoming or outgoing transitions than these. This means that transition systems are total in the sense that in any state at least a d-action is possible. However, afterwards nothing else is possible any more. Thus, taking a d-transition somehow indicates being in a deadlock state.

A *path* in $\mathcal{T}$ is an infinite sequence $\pi = s_0, a_0, s_1, a_1, \dots$, alternating between states and edge labels such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \in \mathbb{N}$. The assumption above ensures that no maximal paths other than infinite ones exist. We write $\Pi_{\mathcal{T}}(s)$ for the set of all paths through $\mathcal{T}$ that start in s. For $i \in \mathbb{N}$, the denote by $\pi \uparrow i$ the ith suffix of π, i.e. $s_i, a_i, s_{i+1}, \dots$.

A path $\pi = s_0, a_0, s_1, a_1, \dots$ determines in a unique way its trace – the ω-word $a_0 a_1 a_2 \dots$ over Σ. Abusing notation we will identify a path with its trace of edge labels and sometimes simply write $\pi \in L$ for a path π and a language L.

2.2. Formal languages and automata

As usual, let Σ be a finite alphabet. Then Σ^ω denotes the set of all infinite words over Σ; ϵ denotes the empty word. A *formal ω-language*, or just language from now on, is a subset of Σ^ω. We are particularly interested in three classes of languages: the ω-regular ones ωREG, the ω-context-free ones ωCFL, and the ω-visibly pushdown ones ωVPL.

In order to be able to use languages in formulas they need to be represented syntactically. Here we choose automata for this purpose: non-deterministic Büchi automata for ωREG [11], non-deterministic Büchi pushdown automata for ωCFL [36], and non-deterministic Büchi visibly pushdown automata for ωVPL [3].

A *non-deterministic Büchi automaton* is a tuple $\mathcal{A} = (Q, q_I, \delta, F)$ where Q is a finite set of *states*, $q_I \in Q$ is a designated *starting state*, $F \subseteq Q$ is a designated set of *acceptance states*, and $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*. A *run* of $\mathcal{A}$ on a word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a sequence of states $q_0, q_1, \dots$ such that $q_0 = q_I$ and $(q_i, a_i, q_{i+1}) \in \delta$ for all $i \in \mathbb{N}$.

A *non-deterministic Büchi pushdown automaton* (ω-PDA) is a tuple $\mathcal{A} = (Q, \Gamma, \bot, q_I, \delta, F)$ where Q, q_I and F are as above, Γ is a finite *stack alphabet*, $\bot \in \Gamma$ is a designated *bottom-of-stack* symbol, and δ is a finite subset of $Q \times \Gamma \times \Sigma \times Q \times \Gamma^*$. A *run* of $\mathcal{A}$ on a word $w = a_0 a_1 \dots$ is a sequence of pairs of states and finite stacks over Γ of the form $(q_0, \gamma_0), (q_1, \gamma_1), \dots$ such that $q_0 = q_I$, $\gamma_0 = \bot$, and for all $i \in \mathbb{N}$: $\gamma_i = \gamma B$, $\gamma_{i+1} = \gamma\gamma'$ and $(q_i, B, a_i, q_{i+1}, \gamma') \in \delta$ for some $B \in \Gamma$ and some $\gamma, \gamma' \in \Gamma^*$.

For the last kind of automaton we need a fixed partition of Σ into three disjoint parts Σ_{push}, Σ_{pop}, and Σ_{int}. A *non-deterministic Büchi visibly pushdown automaton* is a tuple $\mathcal{A} = (Q, \Gamma, \bot, q_I, \delta, F)$ as above with the exception of

$$\delta \subseteq (Q \times \Gamma \times \Sigma_{\mathsf{push}} \times Q \times \Gamma) \cup (Q \times \Gamma \times \Sigma_{\mathsf{pop}} \times Q) \cup (Q \times \Gamma \times \Sigma_{\mathsf{int}} \times Q)$$

A *run* is a sequence of state-stack pairs as above with the provision that for all $i \in \mathbb{N}$ one of the following four cases holds.

- $a_i \in \Sigma_{\mathsf{push}}$, $\gamma_i = \gamma B$, $\gamma_{i+1} = \gamma BC$ for some $\gamma \in \Gamma^*$, $B, C \in \Gamma$, and $(q_i, B, a_i, q_{i+1}, C) \in \delta$.
- $a_i \in \Sigma_{\mathsf{pop}}$, $\gamma_i = \gamma B$ for some $\gamma \in \Gamma^*$, $B \in \Gamma$, $\gamma_{i+1} = \gamma$, and $(q_i, B, a_i, q_{i+1}) \in \delta$.
- $a_i \in \Sigma_{\mathsf{pop}}$, $\gamma_i = \gamma_{i+1} = \bot$, and $(q_i, \bot, a_i, q_{i+1}) \in \delta$.
- $a_i \in \Sigma_{\mathsf{int}}$, $\gamma_i = \gamma B$ for some $\gamma \in \Gamma^*$, $B \in \Gamma$, $\gamma_{i+1} = \gamma_i$ and $(q_i, B, a_i, q_{i+1}) \in \delta$.