Abstract

Model checkers for systems represented by labelled transition systems are not as extensively used as those for systems represented by Kripke structures. This is partially due to the lack of an elegant formal language for property specification which would not be as raw as, for example, HML yet also not as complex as, for example, μ-calculus. This paper proposes a new action-based propositional branching-time temporal logic ACTLW, which enhances popular computation tree logic (CTL) with the notion of actions in a similar but more comprehensive way than action-based CTL introduced by R. De Nicola and F. Vaandrager in 1990. ACTLW is defined by using temporal operators until and unless only, whereas all other temporal operators are derived from them. Fixed-point characterisation of the operators together with symbolic algorithms for global model checking are shown. Usage of this new logic is illustrated by an example of verification of mutual-exclusion algorithms.

Key words: formal verification; model checking; action-based temporal logic; fixed point; mutual-exclusion algorithm

1. Introduction

Model checking is becoming a widely used verification method in both research and commercial projects. Because of many techniques invented to cope with the problem of state-space blow-up, e.g. symbolic algorithms [7] and state-space caching [17], it is definitely not limited to small laboratory examples anymore. An increasing interest in practical applications of model checking is especially well indicated by the rise of user-friendly interfaces for specifying temporal properties [16].
Model checkers presume that a system is specified as a finite-state transition system and its properties are expressed in a suitable formal language. There are two different approaches, either the system being represented by a Kripke structure (KS) or by a labelled transition system (LTS).

In a KS, basic elements are called **atomic propositions** and are assigned to the states, while in an LTS they are called **actions** and are assigned to the transitions. Well-known and widely used languages today, based on LTSs include CCS (e.g. [25]), CSP (e.g. [1]), and LOTOS (e.g. [19] and [26]). Disparate formalisms are used to express the properties of KSs and LTSs. To illustrate the problem one could compare the following two properties:

- If the car is at the crossroad, then its new route will be either East street or West street.
- If the car enters the crossroad, then its next action will be to turn either left or right.

The first property can easily be expressed using state-based formalisms, e.g. temporal logics LTL and CTL [10]. The second one cannot, because it is about the car’s movement (i.e. actions) and not the car’s state (i.e. atomic propositions). We need action-based formalisms for reasoning about actions.

Among different action-based formalisms, action-based temporal logics are a good compromise between expressiveness and clarity. They originate in a simple modal logic called **process logic** introduced in 1985 by M. Hennessy and R. Milner [18]. Today, this formalism is known as Hennessy-Milner logic (HML). HML can provide a characterisation of strong equivalence of LTSs. In 1990, R. De Nicola and F. Vaandrager extended HML and proposed a temporal logic called action-based computation tree logic (ACTL) [23]. Although ACTL imitates CTL syntax, it is still oriented toward adequacy with respect to equivalence relations between LTSs. Hence, an explicit distinction was made between unobservable action $\tau$ and observable actions. In 1993, R. De Nicola et al. provided some additional notes on ACTL and proposed it as a general framework for verifying the properties of action-based models [24]. They also showed how to exploit a CTL model checker for ACTL model checking. The proposed translation function is described in more detail in [12]. Later, ACTL was successfully integrated into several verification tools, e.g. BDD-based ACTL model checker Severo [14] and the XTL prototype model checker, which is part of the CADP toolbox [20]. In both, ACTL slightly differs from the original one, since in Severo unobservable action $\tau$ is treated equally to observable actions and CADP only uses a fragment of ACTL. On the other hand, ACTL model checkers AMC [6] and SAM [13] more strictly follow the original definition of ACTL. Moreover, SAM supports $\mu$-ACTL [11], an extension of ACTL with fixed-point operators. Both, AMC and SAM, are parts of the verification environment JACK, which has been successfully used in a couple of projects, e.g. in verification of a railway signalling system design [4].

Despite the name, ACTL is not a straightforward extension of CTL. Its authors neglected many elegant CTL equivalences and the simplicity of model checking algorithms. This paper introduces a new action-based variant of CTL called action-based CTL with **unless** operator (ACTLW). It can render all ACTL formulae but nevertheless it has analogous model checking algorithms and formulae patterns as CTL. ACTLW is based on temporal operators $\text{until}$ (U) and $\text{unless}$ (W), whereas all other temporal operators are derived from them. In the literature, temporal operator $\text{unless}$ is also known as weak $\text{until}$ and has not been introduced in ACTL.

The paper is further organised as follows. Section 2 gives the syntax and semantics of the new logic. Rules for translation of an ACTL formula into an equivalent ACTLW formula, fixed-point characterisations of ACTLW operators, and useful syntactic abbreviations are also shown. The section is concluded with a discussion about ACTLW semantics over LTSs with deadlocked states. Section 3 presents symbolic algorithms for global ACTLW model checking. In Section 4
we illustrate the use of ACTLW by specifying properties of several mutual-exclusion algorithms. Results from a prototype ACTLW model checker are also given. The paper concludes with a short discussion.

2. Definitions

Action-based computation tree logic with unless operator (ACTLW) is a propositional action-based branching-time temporal logic interpreted over LTSs. It resembles CTL but describes the occurrence of transitions rather than the validity of atomic propositions over time. In contrast to ACTL, ACTLW does not focus on adequacy with respect to equivalence relations but on the simplicity of model checking, i.e. it allows a straightforward implementation due to CTL-style fixed-point characterisations and property specification using common CTL-style equivalences and CTL-style patterns. Formally, an LTS and ACTLW are defined as follows.

Definition 1 (Labelled transition system).
A labelled transition system (LTS) is a quadruple $\mathcal{M} = (\mathcal{S}, \mathcal{A}_\tau, \delta, s_0)$ where:
- $\mathcal{S}$ is a non-empty set of states;
- $\mathcal{A}_\tau$ is a non-empty set of actions containing observable actions and an unobservable action $\tau$;
- $\delta \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ is the transition relation;
- $s_0$ is the initial state.

Set of actions $\mathcal{A}_\tau$ will be called an alphabet of the LTS. In contrast to [23], unobservable action $\tau$ is here defined to be in the alphabet. A triple $(s, \alpha, s') \in \delta$ is called a transition with action $\alpha$ from state $s$ to state $s'$. A path $\pi$ in an LTS is a finite or infinite alternating sequence of states and actions $s_0, a_1, s_1, a_2, s_2, \ldots$ in the LTS such that it starts in a state and for any two successive states $s_{i-1}$ and $s_i$ on the path, $(s_{i-1}, a_i, s_i) \in \delta$. We use notation $\pi(i)$ for identification of particular states on the path $\pi$ (Figure 1) in such a way that $\pi(0)$ is the first state on $\pi$ and $\pi(i+1)$ is the state on the path $\pi$ immediately after the state $\pi(i)$.

![Fig. 1. A path in a labelled transition system](image)

If there exists $(s, \alpha, s') \in \delta$, then state $s'$ is a successor of state $s$. We let $\delta_3(s)$ denote the set of successors of the state $s$ reachable by a transition labelled with an action from the set $A$. A state without successors is called a deadlocked state. Infinite paths and finite paths ending in a state without successors are denoted as fullpaths in an LTS. A state where an infinite path begins such that all actions on this path are unobservable actions $\tau$ is called a divergent state.

Definition 2 (Action-based CTL with unless operator).
Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}_\tau, \delta, s_0)$ be an LTS with a total transition relation. The syntax of ACTLW over $\mathcal{M}$ is defined by the following grammar, where $\alpha \in \mathcal{A}_\tau$ is an arbitrary action, $\mathcal{E}E$ and $\mathcal{A}A$ are path quantifiers, and $U$ and $W$ are until and unless operators, respectively:
The satisfaction of an action formula $\chi$ by an action $a \in A_r$ ($a \models \chi$), a state formula $\varphi$ by a state $s$ ($s \models_M \varphi$), and path formula $\gamma$ by a fullpath $\pi$ ($\pi \models_M \gamma$) is inductively defined by the following rules, where $\alpha \in A_r$ and $/\chi/ = \{ \alpha \mid \alpha \models \chi \}$:

\[
\begin{align*}
\alpha & \models \alpha & \text{iff } a = \alpha \\
\alpha & \models \neg \chi & \text{iff } a \not\models \chi \\
\alpha & \models \chi \lor \chi' & \text{iff } a \models \chi \text{ or } a \models \chi' \\
\chi & \models_M \text{true} & \text{always} \\
\varphi & \models_M \neg \varphi & \text{iff } s \not\models_M \varphi \\
\varphi & \models_M \varphi \lor \varphi' \quad & \text{iff } s \models_M \varphi \text{ or } s \models_M \varphi' \\
\varphi & \models_M \neg \chi & \text{iff there exists a fullpath } \pi \text{ in } M \text{ such that } \pi(0) = s \text{ and } \pi \models_M \gamma \\
\varphi & \models_M \chi & \text{iff for all fullpaths } \pi \text{ in } M: \pi(0) = s \implies \pi \models_M \gamma \\
\varphi & \models_M \chi \land \chi' & \text{iff there exists } \pi(i) \text{ such that } i \geq 1, \pi(i) \models_M \chi', \pi(i) \in \delta_M^i(\pi(i-1)), \text{ and for all } 1 \leq j \leq i-1: \pi(j) \models_M \varphi \text{ and } \pi(j) \in \delta_M^j(\pi(j-1)); \\
\varphi & \models_M \chi \lor \chi' & \text{iff } \pi \models_M \chi \varphi \text{ or } \pi \models_M \chi \lor \chi', \text{ or there does not exist a state } \pi(i) \text{ such that } i \geq 1 \text{ and } \pi(i) \not\models_M \varphi \text{ or } \pi(i) \not\in \delta_M^i(\pi(i-1)).
\end{align*}
\]

We write $\text{true}$ for $\alpha \lor \neg \alpha$ where $\alpha \in A_r$ is some arbitrarily chosen action. In action formulae and state formulae $\text{false}$ stands for $\neg \text{true}$ and various Boolean operators are also used. We will write $\chi \land \chi'$ for $\neg (\neg \chi \lor \neg \chi')$ and $\varphi \land \varphi'$ for $\neg (\neg \varphi \lor \neg \varphi')$. When LTS $M$ is clear from the context, we write $s \models \varphi$ instead of $s \models_M \varphi$. We say that $\varphi$ holds in state $s$ of LTS $M$ iff $s \models_M \varphi$. We say that $\varphi$ holds in LTS $M$ ($M \models \varphi$) iff $\varphi$ holds in the initial state of LTS $M$. Moreover, for action formula $\chi$ we let $/\chi/\!$ denote the set of actions $a \in A_r$ such that $a \models \chi$. For state formula $\varphi$ we let $/\varphi/\!_M$ denote the set of states $p$ in $M = (S, A_r, \delta, s_0)$ such that $p \models_M \varphi$. For $A \subseteq A_r$ and $P \subseteq S$ we let $/A, P/\!_M$ denote the set of transitions $(p, a, p') \in \delta$ such that $a \in A$ and $p' \in P$.

Derived ACTLW operators are introduced in a similar way to CTL (operators using $\text{AA}$ are introduced analogously):

\[
\begin{align*}
\text{EEX}(\chi) \varphi \triangleq \text{EE}[\{\text{false}\} \text{false} \ U \{\chi\} \varphi] \\
\text{EEF}(\chi) \varphi \triangleq \text{EE}[\{\text{true}\} \text{true} \ U \{\chi\} \varphi] \\
\text{EEG}(\chi) \varphi \triangleq \text{EE}[\{\chi\} \varphi \ W \{\text{false}\} \text{false}]
\end{align*}
\]

An informal explanation of ACTLW formulae may be helpful. Let us use term $\varphi$-state to denote state $p$ in $M$ where state formula $\varphi$ holds (i.e. $p \in /\varphi/\!_M$), and term $(\chi, \varphi)$-transition to
denote transition \((p, a, p')\) in \(M\) where action formula \(\chi\) holds for action \(a\) and state formula \(\varphi\) holds in state \(p'\) (i.e. \((p, a, p') \in /\chi/ /\varphi/ /\mathcal{M}\)). Then, the meanings of temporal operators are as follows (Fig. 2):

- \(X\{\chi\} \varphi\) is satisfied on a fullpath iff its first transition is a \((\chi, \varphi)\)-transition.
- \(F\{\chi\} \varphi\) is satisfied on a fullpath iff there exists a \((\chi, \varphi)\)-transition on it.
- \(G\{\chi\} \varphi\) is satisfied on a fullpath iff all transitions on it are \((\chi, \varphi)\)-transitions.
- \([\{\chi\} \varphi \cup \{\chi'\} \varphi']\) is satisfied on a fullpath iff it starts with a finite (possibly empty) sequence of \((\chi, \varphi)\)-transitions followed by a \((\chi', \varphi')\)-transition.
- \([\{\chi\} \varphi \wedge \{\chi'\} \varphi']\) is satisfied on a fullpath iff formula \([\{\chi\} \varphi \cup \{\chi'\} \varphi']\) is satisfied on it or formula \(G\{\chi\} \varphi\) is satisfied on it.

Finally, path quantifiers \(\exists X\) and \(\forall X\) require that the property expressed by the path formula is satisfied for at least one fullpath or for all fullpaths, respectively, starting in the given state. Further, we discuss the expressiveness of ACTLW with regard to ACTL defined in [23]. For every ACTL formula there exists an ACTLW formula which has exactly the same meaning over the same arbitrary LTS. We conjecture that not all ACTLW formulae can be rendered with ACTL (e.g. \(E\{\chi\} \varphi\) if \(\tau \not\models \chi\) and \(\varphi \not\models false\)), but we do not have proof for this.

**Theorem 1.** Let \(M\) be an LTS with a total transition relation. An ACTL formula \(\varphi\) holds in \(M\) iff ACTLW formula \(actl2actlw(\varphi)\) holds in it, where the translation function from ACTL to ACTLW is inductively given as follows:

- \(actl2actlw(true) = true\)
- \(actl2actlw(\neg \varphi) = \neg actl2actlw(\varphi)\)
- \(actl2actlw(\varphi \land \varphi') = actl2actlw(\varphi) \land actl2actlw(\varphi')\)
- \(actl2actlw(\exists X \chi \varphi) = EEX \{\chi \land \neg \tau\} actl2actlw(\varphi)\)
- \(actl2actlw(\forall X \chi \varphi) = AAX \{\chi \land \neg \tau\} actl2actlw(\varphi)\)

Fig. 2. The meaning of temporal operators in ACTLW

\[actl2actlw(\varphi)\]

5
actl2actlw (\(\exists (\varphi, \chi)\) \(\Phi\)) = (actl2actlw (\(\varphi\)) \lor (actl2actlw (\(\varphi\)) \land

\text{EE} [(\(\chi \lor \tau\) actl2actlw (\(\varphi\)) U \(\{\chi \lor \tau\} actl2actlw (\(\varphi\)))]

actl2actlw (\(\forall (\varphi, \chi)\) \(\Phi\)) = (actl2actlw (\(\varphi\)) \lor (actl2actlw (\(\varphi\)) \land

\text{AA} [(\(\chi \lor \tau\) actl2actlw (\(\varphi\)) U \(\{\chi \lor \tau\} actl2actlw (\(\varphi\)))]

\text{Proof.} \text{Follows from the definitions of ACTL and ACTLW.}

As with CTL operators (e.g. [8]), the ACTLW operators can be characterised by fixed-point expressions. We consider elegant to use \(\Phi_3\) and \(\Phi_v\) in the fixed-point expressions which are functions \(2^{\mathcal{S} \times A_r} \times \mathcal{S} \longrightarrow 2^\mathcal{S}\) over \(\mathcal{M} = (\mathcal{S}, A_r, \delta, s_0)\). Let \(T \subseteq \delta\). Then \(p \in \Phi_3(T)\) iff \(\exists a \in A_r \exists p' \in \mathcal{S}\) \((p, a, p') \in \delta \land (p', a, p'') \in T\) and \(p \in \Phi_v(T)\) iff \(\forall a \in A_r \forall p' \in \mathcal{S}\) \((p, a, p') \in \delta \implies (p, a, p'') \in T\). Note that for any \(x, y \subseteq \delta\), \(\Phi_3(x) = \mathcal{S} - \Phi_3(\delta - x)\), \(\Phi_v(x) = \mathcal{S} - \Phi_v(\delta - x)\), \(\Phi_3(x \cup y) = \Phi_3(x) \cup \Phi_3(y)\), \(\Phi_3(x \cap y) = \Phi_3(x) \cap \Phi_3(y)\), and also \(\Phi_v(x \cap y) = \Phi_v(x) \cap \Phi_v(y)\), while in general \(\Phi_v(x \cup y) \neq \Phi_v(x) \cup \Phi_v(y)\).

Let \(\text{lfp}\) and \(\text{gfp}\) denote the least and the greatest fixed points, respectively. ACTLW operators can be characterised by the following expressions:

\[
\begin{align*}
/\text{EEX}\{\chi\} \varphi/M &= \Phi_3(//\chi//, /\varphi/M\//M) \\
/\text{EEF}\{\chi\} \varphi/M &= \text{lfp} \ Z \cdot \Phi_3(//\chi//, /\varphi/M\//M \cup \//A_r, Z//M) \\
/\text{EEG}\{\chi\} \varphi/M &= \text{gfp} \ Z \cdot \Phi_3(//\chi//, /\varphi/M\//M \cap \//A_r, Z//M) = \\
/\text{EE}\{\chi\} \varphi \ W\{\chi'\} \varphi'/M &= \\
\text{lfp} \ Z \cdot \Phi_3(//\chi//, /\varphi/M\//M \cup //\chi//, /\varphi//M \cap Z//M) \\
/\text{EE}\{\chi\} \varphi \ W\{\chi'\} \varphi'/M &= \\
\text{gfp} \ Z \cdot \Phi_3(//\chi//, /\varphi/M\//M \cap //\chi//, /\varphi//M \cap Z//M) \\
/\text{AAX}\{\chi\} \varphi/M &= \Phi_v(//\chi//, /\varphi/M\//M) \\
/\text{AAF}\{\chi\} \varphi/M &= \text{lfp} \ Z \cdot \Phi_v(//\chi//, /\varphi//M\//M \cup \//A_r, Z//M) \\
/\text{AAG}\{\chi\} \varphi/M &= \text{gfp} \ Z \cdot \Phi_v(//\chi//, /\varphi//M\//M \cap \//A_r, Z//M) = \\
/\text{AA}\{\chi\} \varphi \ U\{\chi'\} \varphi'/M &= \\
\text{lfp} \ Z \cdot \Phi_v(//\chi//, /\varphi//M\//M \cup //\chi//, /\varphi//M \cap Z//M) \\
/\text{AA}\{\chi\} \varphi \ W\{\chi'\} \varphi'/M &= \\
\text{gfp} \ Z \cdot \Phi_v(//\chi//, /\varphi//M\//M \cap //\chi//, /\varphi//M \cap Z//M)
\end{align*}
\]
In some cases, a fixed-point characterisation of negated ACTLW operators is helpful. Let us give only expressions which will be further used in the proof of Theorem 2:

\[
\neg \AA \{\chi\} \varphi_M = \\
\gfp Z \cdot \Phi Z(\delta - (\chi / \varphi_M \cup \AA r, (S - Z)) = \\
\gfp Z \cdot \Phi Z((\delta - (\chi / \varphi_M \cup \AA r, (S - Z)) = \\
\gfp Z \cdot \Phi Z(\delta - (\chi \cup \varphi_M \cup \AA r, Z)
\]

Fixed-point characterisations of other negated ACTLW operators can be derived in a similar way, e.g. the characterisation for negation of operator \( \AA \AA \) differs from the characterisation for negation of operator \( \AA \AA \AA \) only by exploiting \( \gfp \) instead of \( \lfp \).

A valuable advantage of CTL over ACTL is the property that all CTL operators can be expressed using only three of them. In fact, different triples can be used. The popular one consists of \( \text{EX}, \text{EG}, \) and \( \text{EU} \). To show that ACTLW allows different bases, we need the following lemma.

**Lemma 1.** Let \( E \) and \( F \) be subsets of a given powerset and \( g \) be a monotonic mapping over subsets of this powerset such that \( g(x \cap y) = g(x) \cup g(y) \) and \( g(x \cap y) = g(x) \cap g(y) \). Then:

\[
\lfp Z(F \cap (E \cap g(Z))) \cup \gfp Z(F \cap g(Z)) = \gfp Z(F \cap (E \cup g(Z)))
\]

**Proof.** Let 0 and 1 denote the empty and universal set, respectively. Let \( k(Z) = F \cap (E \cup g(Z)) \), \( k_u = k(0), k_{u+1} = k(k_u), k_{u+1} = \lfp Z(F \cap (E \cup g(Z))), k_0 = k(1), k_{u+1} = k(k_u) \), and \( h_k = \gfp Z(F \cap (E \cup g(Z))) \). Let \( h(Z) = F \cap g(Z), h_0 = h(1), h_{u+1} = h(h_u) \), and \( h_{u+1} = \gfp Z(F \cap g(Z)) \). Let \( g_{u+1} = g(0), g_{u+1} = g(g_{u+1}), g_{u+1} = g(1) \), and \( g_{u+1} = g(1) \). Let \( n = \max(p, q, r, s, t) \).

\[
k_u = k_{u+1} = k_u \text{ and it can be calculated as follows:}
\]

\[
k_u = F \cap (E \cup g(0)) = (F \cap E) \cup (F \cap g(0))
\]

\[
k_u = F \cap (E \cup g(F \cup g(0))) =
\]

\[
(F \cap E) \cup (F \cap g(F) \cap g(E)) \cup (F \cap g(F) \cap g^2(0))
\]

\[
k_u = (F \cap E) \cup (F \cap g(F) \cap g(E)) \cup \ldots \cup (F \cap g(F) \cap g^n(F) \cap g^n(E)) \cup (F \cap g(F) \cap \ldots \cap g^n(F) \cap g^n(1))
\]

Also \( k_{u+1} = k_{u+1} = k_u \) and it can be similarly calculated as follows:

\[
k_{u+1} = (F \cap E) \cup (F \cap g(F) \cap g(E)) \cup \ldots \cup (F \cap g(F) \cap \ldots \cap g^n(F) \cap g^n(1))
\]

\[
(F \cap g(F) \cap \ldots \cap g^n(F) \cap g^n(1))
\]

7
Finally, $ku_n \cup hv_n = (F \cap E) \cup (F \cap g(F) \cap g(E)) \cup \ldots \cup (F \cap g(F) \cap g^n(F) \cap g^n(E)) \cup (F \cap g(F) \cap \ldots \cap g^n(F) \cap g^n+1(1))$. Because $g^{n+1}(0) = gu_n$, and $g^{n+1}(1) = gu_n$, we can take $g^{n+1}(0) \subseteq g^{n+1}(1)$ and thus $ku_n \cup hv_n = ku_n$. ■

**Theorem 2.**

\[
EE[\{\chi\} \varphi \ W \{\chi'\} \varphi'] = EE[\{\chi\} \varphi \ U \{\chi'\} \varphi'] \lor EEG \{\chi\} \varphi
\]

\[
AA[\{\chi\} \varphi \ U \{\chi'\} \varphi'] = AA[\{\chi\} \varphi \ W \{\chi'\} \varphi'] \land AAF \{\chi'\} \varphi'
\]

**Proof.** This can be proved by using fixed-point characterisations of ACTLW operators and Lemma 1. Let $\mathcal{M} = (S, A, \tau, Z/ /, s_0)$ be an LTS. To prove the first equation we take $F = S$, $E = F_{\Phi}(\|/ /, / /\delta / /, / /\mathcal{M})$, and $g(Z) = F_{\Phi}(\|/ /, / /\tau/ /, / /\mathcal{M})$. To prove the second equation we negate it to obtain $\neg AA[\{\chi\} \varphi \ U \{\chi'\} \varphi'] = \neg AA[\{\chi\} \varphi \ W \{\chi'\} \varphi'] \lor \neg AAF \{\chi'\} \varphi'$ and take $F = F_{\Phi}(\|/ /, / /\delta / /, / /\mathcal{M})$, $E = F_{\Phi}(\|/ /, / /\tau/ /, / /\mathcal{M})$, and $g(Z) = F_{\Phi}(\|/ /, / /\mathcal{M})$. ■

**Consequence.** The set \{EEU, EEG, AAF, AAW\} is an adequate set of ACTLW operators, i.e. they are sufficient to express all other ACTLW operators.

**Conjecture 1.** A minimal set of operators for ACTLW contains four operators. More precisely, it contains operators EEU and AAF, one operator from set \{EEG, EEW\}, and one operator from set \{AAF, AAW\}.

Efficient ACTLW syntactic abbreviations can be achieved by defining \{true\} $\vartriangleq$ true and \{false\} $\vartriangleq$ false. In this way, the following short formulae are obtained:

\[
\begin{align*}
EEX[\{\chi\} \varphi] & \vartriangleq EE[\{\chi\} \varphi] \text{true} \\
EEF[\{\chi\} \varphi] & \vartriangleq EEF[\{\chi\} \varphi] \text{true} \\
EEG[\{\chi\} \varphi] & \vartriangleq EEG[\{\chi\} \varphi] \text{true} \\
EE[\{\chi\} \ U \{\chi'\} \varphi'] & \vartriangleq EE[\{\chi\} \ U \{\chi'\} \varphi'] \\
EE[\{\chi\} \ W \{\chi'\} \varphi'] & \vartriangleq EE[\{\chi\} \ W \{\chi'\} \varphi'] \\
EE[\varphi \ U \varphi'] & \vartriangleq EE[\varphi \ U \varphi'] \\
EE[\varphi \ W \varphi'] & \vartriangleq EE[\varphi \ W \varphi'] \\
EE[\varphi \ U \varphi'] & \vartriangleq EE[\varphi \ U \varphi'] \\
EE[\varphi \ W \varphi'] & \vartriangleq EE[\varphi \ W \varphi']
\end{align*}
\]

Abbreviations of ACTLW formulae with AA operator are analogous to those given above. By using the presented abbreviations, many equivalences given for CTL can be adapted to ACTLW.
Theorem 3.

\[ \text{EEX} \varphi = \neg \text{AAX} \neg \varphi \]
\[ \text{EEF} \varphi = \neg \text{AAG} \neg \varphi \]
\[ \text{EEG} \varphi = \neg \text{AAF} \neg \varphi \]
\[ \text{EE}[\varphi \ W \ \varphi'] = \text{EE}[\varphi \ U \ \varphi'] \lor \text{EEG} \ \varphi \]
\[ \text{AA}[\varphi \ U \ \varphi'] = \text{AA}[\varphi \ W \ \varphi'] \land \text{AAF} \ \varphi' \]
\[ \text{EEX} \{\chi\} = \neg \text{AAX} \{\neg \chi\} \]
\[ \text{EEF} \{\chi\} = \neg \text{AAG} \{\neg \chi\} \]
\[ \text{EEG} \{\chi\} = \neg \text{AAF} \{\neg \chi\} \]
\[ \text{EE}[\{\chi\} \ W \ \{\chi'\}] = \text{EE}[\{\chi\} \ U \ \{\chi'\}] \lor \text{EEG} \{\chi\} \]
\[ \text{AA}[\{\chi\} \ U \ \{\chi'\}] = \text{AA}[\{\chi\} \ W \ \{\chi'\}] \land \text{AAF} \{\chi'\} \]

Proof. These equivalences can be proved by fixed-point characterisations of ACTLW operators.

In the first three cases one should consider that the set of transitions which are not \((true, \varphi)\)-transitions is the same as the set of \((true, \neg \varphi)\)-transitions. The fourth and the fifth equivalence follow directly from Theorem 2. Others are similar.

In Definition 2 there is a requirement for the transition relation of LTS to be total. There is no reason why ACTLW formulae could not be validated over an LTS containing deadlocked states. If deadlocked states are introduced in the LTS, then fixed-point characterisations of ACTLW operators \text{EE}, \text{EEX}, \text{AAF}, and \text{AAU} must be adapted. For example, let \(S_d\) be a set of all deadlock states in LTS \(M\). Then the following fixed-point characterisations of ACTLW operators are in accordance with Definition 3 (the adaptation of others is analogous):

\[
\text{gfp} \ Z : (\Phi_3(//X//,//\varphi//\setminus M) \cup //X//) \cup //\varphi//\setminus M \cup S_d).
\]

\[
\text{lfp} \ Z : (\Phi_3(//X//,//\varphi//\setminus M) \cup //X//) \setminus //\varphi//\setminus M \cup S_d).
\]

After the introduction of deadlocked states, Theorems 1 and 2 remain valid. Furthermore, ACTLW becomes capable of detecting and distinguishing deadlocked and divergent states. The following patterns can be used:

- The state is a deadlocked state: \text{AAX} \{false\}
- There exists a reachable deadlocked state: \(\text{AAX} \{false\} \lor \text{EEF AAX} \{false\}\)
- The state is a divergent state: \(\text{EEX} \{true\} \land \text{EEG} \{\tau\} \land \text{EEX} \{true\}\)
- There exists a reachable divergent state: \(\text{EEF EEG} \{\tau\} \land \text{EEX} \{true\}\)
3. ACTLW model checking

The ACTLW model checking problem is to determine if the given ACTLW formula holds in the given finite-state LTS. Here, we present symbolic algorithms which evaluate formulae with the operators from the ACTLW definition. We write \( cf_S(x) \) to denote the characteristic function of set of states \( S \) encoded with variables \( r_0, r_1, \ldots \), \( cf_A(a) \) to denote the characteristic function of set of actions \( A \) encoded with variables \( a_0, a_1, \ldots \), and \( cf_D(r, a, s) \) to denote the characteristic function of transition relation \( D \) encoded with variables \( r_0, r_1, \ldots, a_0, a_1, \ldots, s_0, s_1, \ldots \).

The set of states in which formula \( EE[\{\chi\} \varphi U \{\chi'\} \varphi'] \) holds can be calculated as:

\[
\begin{align*}
\text{Resolve}_{EEU} & (cf_D(z, a, s), cf_\chi(a), cf_\varphi(z), cf_\chi'(a), cf_\varphi'(z)) \\
& \{
\quad \text{func1} = \text{AND}(cf_\chi, cf_\varphi); \\
\quad \text{func2} = \text{EEX}(cf_D, \text{AND}(cf_\chi', cf_\varphi')); \\
\quad Z = \text{func2}; \\
\quad \text{last} = 1; \\
\quad \text{while} (Z != \text{last}) \\
\quad & \{ \\
\quad & \quad \text{last} = Z; \\
\quad & \quad Z = \text{OR}(\text{func2}, \text{EEX}(cf_D, \text{AND}(\text{func1}, Z))); \\
\quad & \} \\
\quad \text{return} Z;
\}
\end{align*}
\]

Fig. 3. An efficient algorithm for resolving operator \( EEU \)

Similarly, the set of states in which formula \( EE[\{\chi\} \varphi W \{\chi'\} \varphi'] \) holds can be calculated as:

\[
\begin{align*}
\text{func1} & = \text{AND}(cf_\chi, cf_\varphi); \\
\text{func2} & = \text{EEX}(cf_D, \text{AND}(cf_\chi', cf_\varphi')); \\
Z & = \text{func2}; \\
\text{last} & = 1; \\
\text{while} (Z != \text{last}) \\
& \{ \\
& \quad \text{last} = Z; \\
& \quad Z = \text{OR}(\text{func2}, \text{EEX}(cf_D, \text{AND}(\text{func1}, Z))); \\
& \} \\
\text{return} Z;
\end{align*}
\]

The algorithm obtained from this formula is almost identical to the algorithm for resolving operator \( EEU \). The only differences are that the greatest fixed point should be calculated and that the set of deadlocked states should be considered.
The set of states in which formula \( \text{AA}[\{\chi\} \varphi \cup \{\chi'\} \varphi'] \) holds can be calculated as:

\[
/\text{AA}[\{\chi\} \varphi \cup \{\chi'\} \varphi']/\mathcal{M} =
\text{lfp} \ Z \cdot (\Phi_{\varphi}(//\chi/; //\varphi'/\mathcal{M} \cup //\chi/. //\varphi/\mathcal{M} \cap Z//\mathcal{M}) - S_d)
\]

The set of states in which formula \( \text{AA}[\{\chi\} \varphi \mathcal{W} \{\chi'\} \varphi'] \) holds can be calculated as:

\[
/\text{AA}[\{\chi\} \varphi \mathcal{W} \{\chi'\} \varphi']//\mathcal{M} =
\text{gfp} \ Z \cdot \Phi_{\varphi}(//\chi/; //\varphi'/\mathcal{M} \cup //\chi/. //\varphi/\mathcal{M} \cap Z//\mathcal{M})
\]

The algorithm obtained from this formula is almost identical to the algorithm for resolving operator \( \text{AAU} \). The only differences are that the greatest fixed point should be calculated and that the set of deadlocked states does not need to be considered.

4. Example

Mutual-exclusion algorithms are small but quite popular examples of concurrent systems [2,9,27]. They solve the problem of preventing the simultaneous entering of two or more processes into a special part of their code called a critical section. Mutual exclusion has an important role in computer science, as it is, for example, needed to avoid simultaneous writing and reading of the same memory location, which can cause an improper behaviour of the system.

There exist many well-known mutual-exclusion algorithms. We report about the verification of Dekker’s algorithm (1965), Hyman’s algorithm (1966), Peterson’s algorithm (1981), and three variants of Lamport’s Bakery algorithm (1974). In all cases, our system is composed of two processes applying for entry into the critical section. Among the listed ones, only Bakery algorithm could be extended for an arbitrary number of processes. The processes in the given algorithms employ shared variables \( b1 \) and \( b2 \), which they use to express the intention of entering its critical section, and shared variable \( k \) (\( n_i \) in Bakery algorithm), which they use to decide whether they can enter its critical section. In the algorithms, the initial values of shared variables are the following: \( b1 = false, b2 = false, k = 1, n_1 = 0, \) and \( n_2 = 0. \)
4.1. Dekker’s algorithm

Dekker’s algorithm is one of the oldest and also one of the better-known mutual-exclusion algorithms. In Dekker’s algorithm, a process that wants to enter its critical section first sets its variable $b$ to true. Then it checks $b$ from the other process. If it is set to false, then the process may enter the critical section. However, if $b$ from the other process is true, then both processes want to enter their critical sections. In this case, both processes check variable $k$ that determines which process may persist and which one must drop its request. Dekker’s algorithm is given in Fig. 5.

```
PROCESS P1
    while (true) {
        <noncritical section>
        b1=true;
        c1: while (b2==true) do {
            c2: if (k==2) {
                a3: b1=false;
                c3: while (k==2) wait;
                a4: b1=true;
            }
        }
        a5: <critical section>
        a6: k=2;
        a7: b1=false;
    }

PROCESS P2
    while (true) {
        <noncritical section>
        b2=true;
        while (b1==true) {
            if (k==1) {
                b2=false;
                while (k==1) wait;
            }
        }
        <critical section>
        k=1;
        b2=false;
    }
```

Fig. 5. Dekker’s mutual-exclusion algorithm

4.2. Hyman’s algorithm

Hyman’s algorithm given in Fig. 6 is an example of a too simplified and, therefore, incorrect mutual-exclusion algorithm. It resembles Dekker’s algorithm, but the process that wants to enter its critical section does not check variable $b$ after variable $k$ obtains an appropriate value.

```
PROCESS P1
    while (true) {
        <noncritical section>
        b1=true;
        c1: while (k<>1) {
            c2: while (b2==true) wait;
            a3: k=1;
        }
        a4: <critical section>
        a5: b1=false;
    }

PROCESS P2
    while (true) {
        <noncritical section>
        b2=true;
        while (k<>2) {
            c2: while (b1==true) wait;
            a3: k=2;
        }
        <critical section>
        b2=false;
    }
```

Fig. 6. Hyman’s mutual-exclusion algorithm

The problem in Hyman’s algorithm is the following. Let $b1 = false$, $b2 = false$, and $k = 1$, and let both processes be outside the critical section. Suppose that process P2 executes the lines labelled with $a2$, $c1$, and $c2$, and then for some reason stops running before it sets variable $k$ to 2. While P2 is being stopped, process P1 starts. Because the value of variable $k$ is equal to 1, it enters its critical section. Afterwards, process P2 may continue, set variable $k$ to 2, and enter its critical section, too.
4.3. Peterson’s algorithm

Peterson’s algorithm given in Fig. 7 is one of the newer and simpler mutual-exclusion algorithms. Basically, it is again a variant of Dekker’s algorithm. In Peterson’s algorithm, the process that wants to enter its critical section sets its variable \( b \) to true and then it also sets variable \( k \) to point to another process. Thus, each process which wants to enter its critical section first generously gives precedence to the other one when entering the critical section.

```
PROCESS P1
while (true) {
    a1: <noncritical section>
    a2: b1=true;
    a3: k=2;
    c1: while (b2==true and k==2)
        { wait;
        }
    a4: <critical section>
    a5: b1=false;
}
```

```
PROCESS P2
while (true) {
    <noncritical section>
    b2=true;
    k=1;
    while (b1==true and k==1)
        { wait;
        }
    <critical section>
    b2=false;
}
```

Fig. 7. Peterson’s mutual-exclusion algorithm

4.4. Lamport’s Bakery algorithm

Lamport’s Bakery algorithm is a frequently addressed example of mutual-exclusion algorithm. It can be extended to more than two processes and it does not depend on any lower level mutual exclusion. Thus it works correctly even if processes simultaneously read and write shared variables. Bakery algorithm follows the principle of serving customers at a bakery, where the processes are the customers who can be served only one at a time. They determine the order in such a way that each process takes a ticket number. The process with the lowest number is allowed to enter the critical section. Basically, this is an infinite system since ticket numbers may grow arbitrarily large. When a process leaves its critical section, it gets ticket number 0, which indicates that the process is not interested in entering the critical section.

The original Bakery algorithm is given in Fig. 8. Processes are not completely symmetrical, but differ in the condition labelled with \( c2 \). Suppose that both processes are in their initial states and that process P1 wants to enter its critical section. It sets variable \( b1 \) to true and performs addition in the line labelled with \( a3 \). Afterwards, process P1 enters its critical section as it has the ticket with the lowest number and conditions \( c1 \) and \( c2 \) allow this. Now suppose that process P2 also wants to enter the critical section. The value of variable \( n2 \) is set to 2, because the value of \( n1 \) is 1. Thus process P2 is being denied entry to the critical section and must wait. After process P1 leaves the critical section, it gets a ticket with the number 0. Process P2 has now the ticket with the lowest number and may enter the critical section. The situation is even more interesting when processes P1 and P2 want to enter their critical sections simultaneously. In this case, it may happen that variables \( n1 \) and \( n2 \) both get value 1. Due to the different conditions \( c2 \), process P1 will enter the critical section, while process P2 will not do this.

In the case of the system composed of two processes some authors have intentionally or inadvertently simplified the original algorithm. We know about two such simplifications. The first of them (Fig. 9) has been found in [3] and will be called Ben-Ari’s variant after the author of
this book. The second simplification (Fig. 10) will be called the STeP variant [5]. In Ben-Ari’s variant, processes do not use variables \( b_1 \) and \( b_2 \). Their role is compensated by variables \( n_1 \) and \( n_2 \). The STeP variant is even plainer. Variables \( b_1 \) and \( b_2 \) are simply removed from the original algorithm.

```plaintext
PROCESS P1
while (true) {
    a1: <noncritical section>
    a2: b1=true;
    a3: n1=n2+1;
    a4: b1=false;
    c1: while (b2==true) wait;
    c2: while (n2<>0 and n1>n2) {
        wait;
    }
    a5: <critical section>
    a6: n1=0;
}

PROCESS P2
while (true) {
    <noncritical section>
    b2=true;
    n2=n1+1;
    while (b1==true) wait;
    while (n1<>0 and n2>=n1) {
        wait;
    }
    <critical section>
    n2=0;
}
```

Fig. 8. Lamport’s Bakery mutual-exclusion algorithm

```plaintext
PROCESS P1
while (true) {
    a1: <noncritical section>
    a2: n1=1;
    a3: n1=n2+1;
    c2: while (n2<>0 and n1>n2) {
        wait;
    }
    a5: <critical section>
    a6: n1=0;
}

PROCESS P2
while (true) {
    <noncritical section>
    n2=1;
    n2=n1+1;
    while (n1<>0 and n2>=n1) {
        wait;
    }
    <critical section>
    n2=0;
}
```

Fig. 9. Ben-Ari’s variant of Bakery mutual-exclusion algorithm

```plaintext
PROCESS P1
while (true) {
    a1: <noncritical section>
    a3: n1=n2+1;
    c2: while (n2<>0 and n1>n2) {
        wait;
    }
    a5: <critical section>
    a6: n1=0;
}

PROCESS P2
while (true) {
    <noncritical section>
    n2=n1+1;
    while (n1<>0 and n2>=n1) {
        wait;
    }
    <critical section>
    n2=0;
}
```

Fig. 10. Bakery mutual-exclusion algorithm from project STeP

It turns out that the the STeP variant is only conditionally correct. More precisely, it relies on atomic implementation of the incrementation. In our model, the incrementation is implemented as a two-step operation, i.e. first read the old value and then write the new value, which leads to an invalid situation, where due to the simultaneous incrementation of variables \( n_1 \) and \( n_2 \) both processes get the same number and enter the critical section at the same time.
4.5. The verification

Verification of the mutual-exclusion algorithms was done with a prototype ACTLW model checker, which we have implemented as a module of open-source tool EST, 2nd Ed. (http://lms.uni-mb.si/EST/). EST is a BDD-based tool for verification of concurrent systems. It includes an extended CCS parser, runs on different platforms (including Linux and Microsoft Windows), and distinguishes itself with an easily readable source code written in C.

A system to be verified with EST is represented as a parallel composition of LTSs [22]. For example, the set of LTSs for the STeP variant of Bakery algorithm consists of $N_1, N_2, N_1INC, N_2INC, P_1,$ and $P_2$. LTSs $N_1$ and $N_1INC$ are shown in Fig. 11. They are used to store and to increment the value of variable $n_1$. LTSs $N_2$ and $N_2INC$ are analogous. LTSs $P_1$ and $P_2$ (shown in Fig. 12) represent the algorithm of processes $P_1$ and $P_2$, respectively.

---

---

Fig. 11. LTS $N_1$ representing variable $n_1$ and LTS $N_1INC$ used to increment it.

---

---

Fig. 12. Models of processes $P_1$ and $P_2$ in the STeP variant of Bakery algorithm.
Writing, comparing and the incrementation of variable \( n1 \) is performed via actions \( \text{l}n1\text{w}0 \),
\( \text{l}n1\text{w}1 \), \( \text{l}n1\text{w}2 \) (write value 0, 1, or 2), \( ?n1\text{r}0 \), \( ?n1\text{r}1 \), \( ?n1\text{r}2 \) (read the value and proceed if it
is equal to 0, 1, or 2), and \( \text{l}n1\text{inc} \) (increment the value by 1). In such a manner, only variables
with a finite (and for practical reasons small enough) domain can be modelled. This turns out
to be a limitation of the Bakery algorithm, where processes use shared variables whose domains
are integer numbers without any upper bound. In our model, if the values of variables \( n1 \) or \( n2 \)
exceeds 2, the model reaches a deadlocked state, which is a warning that the domain is too small.

To facilitate verification, a transition with action \( !\text{request1} \) or \( !\text{request2} \), respectively,
has been added to all the models to obtain notification that a particular process applied for entry
to its critical section. For example, in the STeP variant of Bakery algorithm a transition with
action \( !\text{request1} \) is inserted in LTS \( P1 \) just after transition \( s1 \rightarrow ?\text{n1inc} \rightarrow s2 \).

Below we give properties checked with the ACTLW model checker. To illustrate the difference
between ACTLW and ACTL, we also show how these properties can be specified with ACTL.

Please, note that we shortened some formulae by using (a strong version of) Hennessy-Milner’s
operator [], which is in ACTLW and ACTL defined as \( \neg\text{E}\text{EX}\{\chi\} \) \( (\neg\chi) \) and \( \neg\exists\chi \neg\phi \), respectively.
Moreover, to be able to give ACTL formulae \( F1 \) and \( F2 \), we omit the requirement for the
transition relation from the definition of ACTL to be total.

(i) [Deadlock] The composed model contains a deadlocked state.

\[
F1 = (\text{AAX} \{\text{false}\}) \lor (\text{EEF AAX} \{\text{false}\})
\]

A corresponding ACTL formula: \( F1 = \exists F (\neg\exists X_{\text{true},\text{true}}) \land (\neg\exists X_{\text{true},\text{true}}) \)

(ii) [Divergent] The composed model contains a divergent state.

\[
F2 = \text{EEF EEG} \{\tau\} \text{EEG} \{\text{true}\}
\]

A corresponding ACTL formula: \( F2 = \exists F \neg\forall (true \land true \land true) \)

(iii) [Mutual exclusion] If process \( P1 \) (\( P2 \)) enters the critical section, then process \( P2 \) (\( P1 \))
cannot enter it, unless process \( P1 \) (\( P2 \)) leaves the critical section.

\[
F3a = \text{AAG}[\text{enter1}] \text{AA}[\neg\text{enter2}] W[\text{exit1}]
\]

\[
F3b = \text{AAG}[\text{enter2}] \text{AA}[\neg\text{enter1}] W[\text{exit2}]
\]

A corresponding ACTL formula: \( F3a = \forall G[\text{enter1}] \neg\exists (true \land \neg\text{exit1} \land \text{enter1} \land \text{true}) \)

(iv) [Overtaking] If process \( P1 \) (\( P2 \)) wants to enter the critical section, then process \( P2 \) (\( P1 \))
can enter the critical section at most once before process \( P1 \) (\( P2 \)) enters the critical section.

\[
F4a = \text{AAG}[\text{request1}] \neg\text{EE}[\neg\text{enter1}] U[\text{enter2} \land \text{EE}[\neg\text{enter1} U[\text{enter1}]]]
\]

\[
F4b = \text{AAG}[\text{request2}] \neg\text{EE}[\neg\text{enter2}] U[\text{enter1} \land \text{EE}[\neg\text{enter2} U[\text{enter1}]]]
\]

A corresponding ACTL formula:

\[
F4a = \forall G[\text{request1}] \neg\exists (true \land \neg\text{enter1} \land \text{enter1} \land \exists (true \land \neg\text{enter2} \land \text{enter2})
\]

Due to the special meaning of unobservable action \( \tau \), ACTL formula \( F2 \) does not distinguish
deadlocked and divergent states. Hence, it corresponds to the given ACTLW formula only for
LTSs without deadlocked states. Formulae \( F3 \) and \( F4 \) show the usefulness of operator \( W \) and
ACTLW formulae abbreviations. ACTL formulae corresponding to \( F3b \) and \( F4b \) are analogous
to \( F3a \) and \( F4a \), respectively.

Although performance analysis of our ACTLW model checker is not the goal of this paper,
we report the size of the composed models (Fig. 13). The verification of every algorithm was finished within seconds on a modern computer.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of states</th>
<th>Number of transitions</th>
<th>Number of transitions with observable action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker’s algorithm</td>
<td>164</td>
<td>310</td>
<td>88</td>
</tr>
<tr>
<td>Hyman’s algorithm</td>
<td>96</td>
<td>182</td>
<td>84</td>
</tr>
<tr>
<td>Peterson’s algorithm</td>
<td>68</td>
<td>136</td>
<td>42</td>
</tr>
<tr>
<td>Lamport’s algorithm</td>
<td>292</td>
<td>521</td>
<td>121</td>
</tr>
<tr>
<td>Ben-Ari’s variant</td>
<td>214</td>
<td>383</td>
<td>102</td>
</tr>
<tr>
<td>The STeP variant</td>
<td>242</td>
<td>451</td>
<td>162</td>
</tr>
</tbody>
</table>

Fig. 13. Size of models of mutual-exclusion algorithms

The table in Fig. 14 gives verification results. It turns out that the model of Lamport’s variant and also the model of Ben-Ari’s variant of Bakery algorithm contain deadlocked states. They are a consequence of the bounded domain of variables \( n_1 \) and \( n_2 \). Namely, Bakery algorithm presumes an unlimited set of values for these variables, whereas the model is built in such a way that whenever the value of \( n_1 \) or \( n_2 \) is equal to the maximal allowed value \( MAX \) and the incrementation is demanded, the model reaches a deadlocked state. We have checked that by defining \( MAX + 1 = 0 \), \( MAX + 1 = 1 \), or \( MAX + 1 = MAX \), the algorithms work incorrectly and hence, the unbounded domain is an essential requirement for both variants.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Deadlock</th>
<th>Divergent</th>
<th>Mutual exclusion</th>
<th>Overtaking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dekker’s algorithm</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Hyman’s algorithm</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Peterson’s algorithm</td>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>Lamport’s algorithm</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>Ben-Ari’s variant</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>The STeP variant</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

Fig. 14. Results of mutual-exclusion algorithms verification

Verification also pointed out that some models contain divergent states. Infinite sequences of unobservable actions \( \tau \) are a consequence of modelling and not the property of the algorithms. Divergent states do not influence the results of model checking regarding mutual exclusion and overtaking, but they complicate the checking of liveness properties. Indeed, fairness constraints are needed to check liveness properties even if the model does not contain divergent states [9]. We have not implemented fairness constraints, yet, and we refer a curious reader to see e.g. [15].
5. Conclusion

LTSs can be used to model the behavioural aspects of concurrent and distributed systems, e.g. communications protocols, mutual-exclusion algorithms, object-oriented software, asynchronous circuits etc. On the other hand, model checking is one of the most promising verification techniques drawing a lot of attention to its results when used for reasoning over Kripke structures. The fusion of these two concepts leads to model checkers focusing on actions rather than states and inspires the development of action-based temporal logics.

This paper introduces ACTLW, a propositional branching-time temporal logic similar to CTL, which describes the occurrence of actions over time. It can be considered as an enriched variant of ACTL introduced by R. De Nicola and F. Vaandrager in [23]. All properties expressed with ACTL formulae can be expressed with ACTLW formulae. Compared to ACTL, the definition of ACTLW contains the additional operator unless (W), whereas the path operators $\psi^*_\psi', X\tau$, and $X\chi$ are absent. Moreover, ACTLW allows and does not impose the usage of unobservable action $\tau$ for expressing the properties. This gives flexibility and also contributes to the expressiveness of ACTLW formulae. Moreover, the syntax of ACTLW allows constant true to be omitted from the formulae, which results in a framework where the properties can be expressed with patterns similar to those used with CTL [21]. For example, ACTLW formula $\text{EEF EEG }\{\tau\} \text{ EEX }\{\text{true}\}$ is quite a comprehensive way of expressing that there exists a reachable divergent state. For comparison, the same property is expressed in ACTL using a definitely more ambiguous formula $\exists F \neg \forall (\text{true true} U \text{true true})$, which even has a drawback that it is inappropriate for LTSs with deadlocked states.

We proposed notation $\text{EE}$ and $\text{AA}$ for path quantifiers in ACTLW to distinguish ACTL and ACTLW formulae in plain text format. In this way, both types of operators can coexist and the compatibility with existing ACTL model checkers is preserved. This notation even allows mixing of ACTL and ACTLW formulae, e.g. $E[true \{a\} U EEF \{b\} \text{true}] = \exists (true a U EEF \{b\} \text{true})$.

The given symbolic algorithms for ACTLW model checking are similar to those used for CTL model checking [8]. They are based on calculating the least and greatest fixed points. The maximum number of iterations in each algorithm is bounded by the number of states in the LTS, thus the complexity of the algorithms is linear in the number of states in the LTS. The algorithms are also linear in the length of the formula. However, as usual with all global model checking algorithms, an efficient representation of Boolean functions (such as BDDs) is indispensable to avoid the state explosion problem.

The presented algorithms can be extended to generate diagnostics, either in the form of linear witnesses and counterexamples or witness and counterexample automata [21]. Actually, we have implemented an ACTLW model checker with the ability of diagnostics generation and included it into the EST tool. The ACTLW model checker from EST can also be tested online using Witness and Counterexample Server at http://fmt.isti.cnr.it/WCS/.

There are many directions for extending the presented work. We give some theorems about the properties of ACTLW, but there are several others to prove. In particular, we are looking for proof of Conjecture 1 about a minimal adequate set of operators for ACTLW. Other very interesting topics are the relationship between ACTLW and ACTL, especially the differences in their expressiveness, on-the-fly ACTLW model checking, the extension of ACTLW with fairness constraints, and the usage of ACTLW in different interfaces for specifying properties.
References


