Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 125–141
www.elsevier.com/locate/entcs

Local Module Checking for CTL Specifications

Samik Basu[1]

Department of Computer Science
Iowa State University
Ames, USA

Partha S Roop, Roopak Sinha[2,3]

Department of Electical and Computer Engineering
University of Auckland
Auckland, New Zealand

Abstract

Model checking is a well known technique for the verification of finite state models using temporal logic specification. While model checking is suitable for transformational systems (also called closed systems), it is unsuitable for *open systems* (also known as reactive systems) where the nondeterminism in the environment must be considered during verification. *Module checking* is an approach for the verification of open systems which have both closed (internal) and open (environment or external) states. It has been demonstrated in [10] that the complexity of module checking branching time logic CTL is EXPTIME-complete. The approach to module checking is global and the method tries to establish that the property in question holds over all possible environments.
This papers develops a local approach to CTL module checking using tableau rules. The proposed approach tries to determine a single environment under which the negation of the property is satisfied over the given module. Such a strategy, thus, leads to a local approach to module checking where we only explore states that are relevant to proving that the negation of the property can be satisfied over the given module using an appropriate witness (environment) that the algorithm also generates. While the worst case complexity of our algorithm is identical to the earlier complexity, we demonstrate that practical implementation of the proposed approach is feasible and yields much better results than the global approach.

Keywords: module checking, open systems, tableau based verification

1 Introduction

Reactive systems [7,13] are open systems that continuously interact with their environment while executing a non-terminating control program. Examples include

[1] Email: sbasu@cs.iastate.edu
[2] Email: p.roop@auckland.ac.nz
[3] Email: rsin077@ec.auckland.ac.nz

doi:10.1016/j.entcs.2006.02.035

operating systems, communication protocols, missile and avionics systems and controllers for nuclear plants and simple home appliances such as microwaves, DVD players and washing machines. Such applications require careful analysis, design and validation techniques as they can often be safety critical. Hence, *formal techniques* have often been used in the design, development and validation of these systems [16,11,5,6,3,12,15]. Formal techniques use precise syntax and semantics for defining specifications and models of systems so that rigorous verification of properties such as correctness, reliability and security is made possible.

With the advent and widespread use of *embedded systems*, which are ubiquitous reactive computing systems ranging from simple home appliances to very complex applications in avionics and defense, the need for formal methods in the design of reactive systems is growing. One very common approach to verification of reactive systems has been *model checking*. A model checker takes a formula in some *temporal logic* [14] as a desirable property and performs formal analysis over a finite-state model of a system (called a *Kripke structure* [3], a special class of finite state machines). The model checking process is essentially an automated reachability analysis task over the finite state model of the system. This task either terminates with a proof that the temporal property holds over the model or on failure generates a counter example.

The model checker, during the reachability analysis phase, assumes that all transitions out of any given state of a model is eventually enabled. This assumption is based on the fact that the model is considered to be *closed* i.e., all states of the system are purely internal and that it is the system that gets to choose which transition to take based on some internal computation. This approach to analysis, while being suitable for *transformational systems* which are closed, are unsuitable for reactive systems that are *open*.

An open system maintains an ongoing interaction with its environment. Hence, the state-space of such a system may be partitioned into a set of states that are open (also called external or *environment* states) and another set of states that are closed (also called internal or *system* states) [8,9]. An environment state reacts to events in the external environment of the open system and the environment is considered asynchronous and uncontrollable. A system state, on the other hand, takes no inputs from the external environment and the system automatically chooses one of the transitions based on some internal decision (such as say the value of a variable or the result returned by a function). Kupferman et al. have recently shown that model checking may not be enough for open systems due to the presence of environment states and when branching formulas are considered in the specification [9]. The proposed technique, called *module checking*, takes the asynchronous environment into account while doing a proof for branching-time logics. [6,5] further discuss techniques devoted to issues concerning verification of open systems.

This paper illustrates the need for module checking reactive systems and proposes an alternate approach to module checking that has efficient implementation avenues compared to the original approach. We first illustrate the need for module checking followed by our approach to *local module checking*.

1.1 Motivating Example - The Coffee Brewer Verification Problem

Consider a simple *coffee brewer* model as depicted in the Figure 1. In this figure, environment states are shown using elipses whereas system states are drawn as circles. Each transition out of any state is marked by a natural number starting with 1. The brewer serves either *five* or *ten* cups of coffee in either *medium* or *strong* flavor. A user can select the *number of cups* of coffee (using a switch as an input) and the *strength of coffee* (using another switch). The brewer is normally in the *off* state until the switched on. Hence, the initial state labeled by the proposition OFF is an environment state. Once the brewer is switched on, it enters a state labeled by the proposition $CHOOSE$. This is again an environment state. In this state, the user can select the number of cups of coffee and the strength of coffee. Depending on the selection (two switches lead to four different possibilities), the brewer enters any one of the following states: $(five, medium)$, $(five, strong)$, $(ten, medium)$ or $(ten, strong)$. Once the selections have been made and the corresponding state has been reached, the *brew cycle switch* has to be switched on to start the brewing. Hence all the above four states are also environment states. Once the brew cycle is set, the brewer makes a transition to a state labeled $BREW$. This state is a system state since no inputs from the environment is required to make progress. From the $BREW$ state, the brewer makes an automatic transition to the $DONE$ state after a predetermined time period (which is the amount of time taken by the brewing process). The state labeled $DONE$ is also a system state since the brewer takes one of two possible branches based on some internal condition. If any error is detected (say not enough coffee or no milk power), then a transition is made to an error state (labeled by $ERROR$). Alternatively, the brewer can reach a state labeled $SERVE$ where the actual coffee selected is served.

CTL is a branching time temporal logic that has been shown to be quite efficient for model checking. Let us consider the following CTL property:

$AGEF\,(TEN \wedge AF\,(SERVE \vee ERROR))$ which demands that *from any state one can possibly eventually select ten cups of coffee and once selected, ten cups will always be served (or an error encountered) in the future.*

Note that a model checker will always return a true answer for this question. However, consider the following situation. Due to cost cutting in the work place where the brewer is installed, brewing ten cups of coffee at a time is not allowed (this has been enforced through a circular and the ten cups switch is masked). Hence, no user will be allowed to make this selection from the $CHOOSE$ state. Thus, it will not be possible to guarantee the selection of ten cups of coffee and hence the property fails to hold over the coffee brewer model. This property could also be violated if all users request five cups of coffee or tea (and no users request ten cups of any beverage). In this case, even though the machine is capable of dispensing ten cups of tea or coffee, the environment in which it operates effectively disables it from doing that.

This property illustrates that due to the presence of environment states, it may not be always possible to satisfy branching time temporal properties. As the environment of an open system is asynchronous and hence uncontrollable, the environ-

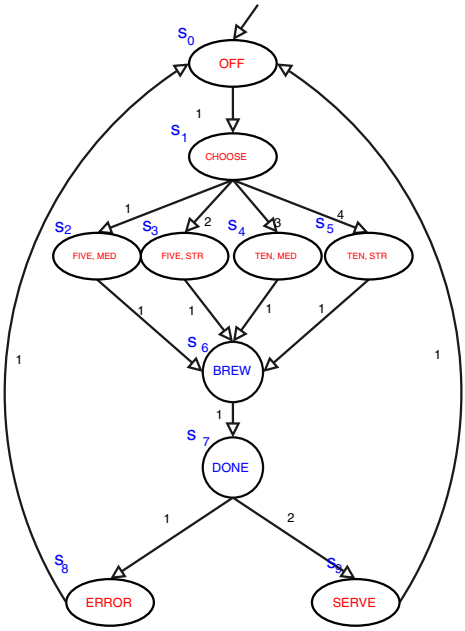


Fig. 1. The coffee brewer example

ment may never enable some desired transitions leading to failure of the property. This example thus motivates the module checking problem: how do we ensure that a given property holds over a known *module* (a model with environment and system states) assuming an uncertain environment. The basic idea in module checking is to prove that the property holds over all possible choices made in the environment states. In other words, the module checking problem is to decide if a CTL formula φ holds over a module $\mathcal{M}$ provided for all possible environments $\mathcal{E}$ such that $\mathcal{M} \times \mathcal{E}$ satisfy φ. Here, $\mathcal{M} \times \mathcal{E}$ denotes the composition of the model and the environment running in parallel (to be formalized later).

This may be expressed as follows: module checking, denoted by $\mathcal{M} \models_o \varphi$, therefore, amounts to deciding whether $\forall \mathcal{E}.\mathcal{M} \times \mathcal{E} \models \varphi$; i.e.

$$\mathcal{M} \models_o \varphi \Leftrightarrow \forall \mathcal{E}.\mathcal{M} \times \mathcal{E} \models \varphi \tag{1}$$

It has been shown in Kupferman et al. [10] that the module checking problem for the temporal logic CTL is EXPTIME-complete unlike the polynomial complexity for the model checking problem. The reason for this complexity may be intuitively seen from the above equation since the proof has to be carried out for all possible environments. This is unlike model checking, where the system is closed and hence the reachability proof proceeds without considering any nondeterminism in the environment.

While this sounds like bad news, a practical implementation of module checking is possible by the following observation.

Proceeding further, from Equation 1 we state that

$$\mathcal{M} \not\models_o \varphi \;\Leftrightarrow\; \exists \mathcal{E}'.(\mathcal{M} \times \mathcal{E}' \not\models \varphi \Leftrightarrow \mathcal{M} \times \mathcal{E}' \models \neg\varphi) \tag{2}$$

In the above, the negation on φ can be pushed inside the formula such that all temporal and boolean operators are free from negation. Furthermore as the existence of $\mathcal{E}'$ ensures that $\mathcal{M} \not\models_o \varphi$, $\mathcal{E}'$ acts as a *witness* to the violation. Such a witness can be used to provide useful insights to the reason for violation of a desired property over an open systems. The main beauty of this procedure is that we are looking for only *one* environment using which we can prove that the formula is not satisfied. We will illustrate later that we can employ a set of tableau rules to perform the computation of $\mathcal{E}'$ locally. This local computation is similar to on-the-fly model checking [1] where the state space of the system under verification is constructed on a need-driven basis. Even though the worst case complexity of local module checking is still bounded by the results of [10], we demonstrate that the proposed approach to local module checking yields much better practical results. The main contributions of this paper are:

(i) We propose a set of sound and complete tableau rules for local module checking. The proposed approach determines a single witness (environment) so that under the witness the negation of the given CTL property is satisfied. The proof proceeds only along a local set of states that are needed for the generation of a witness.

(ii) We have developed a local module checker by extending NuSMV [2]. Local module checking has been compared with the generation of the maximal environment under which the given CTL property is satisfied to demonstrate the performance gain of the proposed approach. The benchmarks are examples from NuSMV with both environment and system states.

The rest of this paper is organized as follows. Section 2 presents fundamental theory. Section 3 formulates CTL local module checking and presents the tableau rules for both system and environment states. Section 4 describes the implementation of a local module checker and section 5 contains the results obtained from it. Concluding remarks follow in section 6.

2 Preliminaries

Kripke Structure.

System behavior is described using Kripke structure $M = (S, s^0, \rightarrow, P, L)$, where S is the set of states, $s^0 \in S$ is the start state, $\rightarrow \subseteq S \times S$ is the set of transition relations, P is the set of propositions relevant to M and finally, $L : S \rightarrow 2^P$ is the labeling function mapping each state to set of propositions.

Temporal logic: CTL.

Properties of the system are defined using branching time temporal logic CTL. A CTL formula is defined over a set of propositions using temporal and boolean operators as follows:

$$\phi \rightarrow P \mid \neg P \mid tt \mid ff \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathtt{AX}\phi \mid \mathtt{EX}\phi \mid \mathtt{A}(\phi \mathtt{U} \phi) \mid \mathtt{E}(\phi \mathtt{U} \phi) \mid \mathtt{AG}\phi \mid \mathtt{EG}\phi$$

Note that CTL in general allows negations on temporal and boolean operators. However, we restrict ourselves to verifying CTL formulas where negations can only be applied to propositions.

Semantics of CTL formula, φ denoted by $[\![\varphi]\!]_M$ is given in terms of set of states in Kripke structure, M, which satisfies the formula. See Fig. 2.

$$
\begin{aligned}
[\![p]\!]_M &= \{s \mid p \in L(s)\} \\
[\![\neg p]\!]_M &= \{s \mid p \notin L(s)\} \\
[\![tt]\!]_M &= S \\
[\![ff]\!]_M &= \Phi \\
[\![\varphi \wedge \psi]\!]_M &= [\![\varphi]\!]_M \cap [\![\psi]\!]_M \\
[\![\varphi \vee \psi]\!]_M &= [\![\varphi]\!]_M \cup [\![\psi]\!]_M \\
[\![AX\varphi]\!]_M &= \{s \mid \forall s \rightarrow s' \wedge s' \in [\![\varphi]\!]_M\} \\
[\![EX\varphi]\!]_M &= \{s \mid \exists s \rightarrow s' \wedge s' \in [\![\varphi]\!]_M\} \\
[\![A(\varphi U \psi)]\!]_M &= \{s \mid \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \wedge \exists i, j.s_j \models \psi \wedge \forall i < j.s_i \models \varphi\} \\
[\![E(\varphi U \psi)]\!]_M &= \{s \mid \exists s = s_1 \rightarrow s_2 \rightarrow \ldots \wedge \exists i, j.s_j \models \psi \wedge \forall i < j.s_i \models \varphi\} \\
[\![AG\varphi]\!]_M &= \{s \mid \forall s = s_1 \rightarrow s_2 \rightarrow \ldots \wedge \forall i.s_i \models \varphi\} \\
[\![EG\varphi]\!]_M &= \{s \mid \exists s = s_1 \rightarrow s_2 \rightarrow \ldots \wedge \forall i.s_i \models \varphi\}
\end{aligned}
$$

Fig. 2. Semantics of CTL

A state $s \in S$ is said to satisfy a CTL formula φ, denoted by $M, s \models \varphi$, if $s \in [\![\varphi]\!]_M$. We will omit M from $\models$ relation and $[\![]\!]$ if the model is evident in the context. We will also say that $M \models \varphi$ iff $M, s^0 \models \varphi$. The complexity for model checking M against a CTL formula φ is $O(|M| \times |\varphi|)$ where $|M|$ and $|\varphi|$ are size of the model and the formula respectively.

Module Checking. [8]

In contrast to model checking where all transitions in every state of the model are always enabled, module checking is specifically directed for verification of models of open systems with states where the environment decides which transitions are enabled. Typically, in models of open systems, modules in short, the states are partitioned into two sets S_s and S_e where S_s consists of system states with *all* outgoing transitions enabled while S_e is the set of environment-controlled states where *some* (at least one) transitions are enabled. Note that, the environment can enable one or more transitions but cannot disable all transitions.

[10] presents the complexities for module checking in the setting of different temporal logic. While LTL module checking problem has the same complexity as LTL model checking, module checking branching-time temporal logics (CTL, CTL*)

is harder compared to corresponding model checking. It has been proved that the problem of module checking is EXPTIME-Complete for CTL and 2EXPTIME-complete for CTL* specifications.

3 Tableau-based CTL Module Checking

In this paper, we present a technique for module checking CTL specifications such that the state-space of the module is explored locally and on-the-fly, i.e. our technique only explores the states needed to (dis)satisfy a given CTL formula.

We consider the behavior of a module $\mathcal{M}$ in the context of an environment $\mathcal{E}$ such that at each system state of $\mathcal{M}$, the environment is incapable of altering the behavior of the module while at each environment state, the environment can decide which transitions to enable. To address such restrictions on the interaction, the behavioral patterns of $\mathcal{M}$ and $\mathcal{E}$ are described using *labeled Kripke structure* defined as follows:

Definition 3.1 [Labeled Kripke Structure] A labeled Kripke structure $LKS = (S, s^0, \rightarrow, P, L, K)$ where S, s^0, P, L are defined as before, K is the maximum out-going branching factor of states in S and $\rightarrow \subseteq S \times \{1, 2, \ldots, n\} \times S$ with $n \leq K$.

The state set S of a module may be partitioned into two subsets, S^e, the set of all environment states and S^s, the set of all system states. In the above, each transition is annotated by a branching identifier whose domain is equal to the maximum branching factor. We will write $s \xrightarrow{i} s'$ to denote the i-th out-going transition from s if $(s, i, s') \in \rightarrow$.

Definition 3.2 [Parallel Composition] Given a module $\mathcal{M} = (S_\mathcal{M}, s^0_\mathcal{M}, \rightarrow_\mathcal{M}, P_\mathcal{M}, L_\mathcal{M}, K)$, its environment $\mathcal{E} = (S_\mathcal{E}, s^0_\mathcal{E}, \rightarrow_\mathcal{E}, P_\mathcal{E}, L_\mathcal{E}, K)$, their parallel composition resulting in $\mathcal{M} \times \mathcal{E} \equiv \mathcal{P} = (S_\mathcal{P}, s^0_\mathcal{P}, \rightarrow_\mathcal{P}, P_\mathcal{P}, L_\mathcal{P}, K)$ are defined as follows

(i) $S_\mathcal{P} \subseteq S_\mathcal{M} \times S_\mathcal{E}$

(ii) $s^0_\mathcal{P} = (s^0_\mathcal{M}, s^0_\mathcal{E})$

(iii) $P_\mathcal{P} = P_\mathcal{M} \cup P_\mathcal{E}$

(iv) $L_\mathcal{P}(s_1, s_2) = L_\mathcal{M}(s_1) \cup L_\mathcal{E}(s_2)$ where $s_1 \in S_\mathcal{M}$ and $s_2 \in S_\mathcal{E}$

(v) $(s_1, s_2) \xrightarrow{i}_\mathcal{P} (s'_1, s'_2)$ if $s_1 \xrightarrow{i}_\mathcal{M} s'_1$ and $s_2 \xrightarrow{i}_\mathcal{E} s'_2$ where $s_1, s'_1 \in S_\mathcal{M}$ and $s_2, s'_2 \in S_\mathcal{E}$
Furthermore, following constraints are imposed to restrict $\mathcal{E}$
   (a) System-state conformity. if s_1 is a system state then $\forall s_1 \xrightarrow{i}_\mathcal{M} s'_1 \Rightarrow \exists s_2 \xrightarrow{i}_\mathcal{E} s'_2$ and vice versa.
   (b) Environment-controllability. if s_1 is an environment-controlled state then $\exists s_2 \xrightarrow{i}_\mathcal{E} s'_2 \wedge (\forall s_2 \xrightarrow{j}_\mathcal{E} s''_2 \Rightarrow \exists s_1 \xrightarrow{j}_\mathcal{M} s''_1)$

Given a CTL formula φ, we say that $\mathcal{M} \times \mathcal{E} \equiv \mathcal{P} \models \varphi \Leftrightarrow s^0_\mathcal{P} \models \varphi$. Module checking, denoted by $\mathcal{M} \models_o \varphi$, therefore, amounts to deciding whether $\forall \mathcal{E}. \mathcal{M} \times \mathcal{E} \models$

$$\texttt{reorg}\ \frac{s_s|e \models \{\varphi_1,\ldots,\varphi_n\}}{s_s|e \models \varphi_1\ \ldots\ s_s|e \models \varphi_n}$$

$$\texttt{prop}\ \frac{s_s|e \models p}{.}\ p \in L(s_s) \qquad \wedge\ \frac{s_s|e \models \varphi_1 \wedge \varphi_2}{s_s|e \models \varphi_1 \quad s_s|e \models \varphi_2}$$

$$\vee_1\ \frac{s_s|e \models \varphi_1 \vee \varphi_2}{s_s|e \models \varphi_1} \qquad \vee_2\ \frac{s_s|e \models \varphi_1 \vee \varphi_2}{s_s|e \models \varphi_2}$$

$$\texttt{unr}_{eu}\ \frac{s_s|e \models E(\varphi U \psi)}{s_s|e \models \psi \vee (\varphi \wedge EXE(\varphi U \psi))} \qquad \texttt{unr}_{au}\ \frac{s_s|e \models A(\varphi U \psi)}{s_s|e \models \psi \vee (\varphi \wedge AXA(\varphi U \psi))}$$

$$\texttt{unr}_{eg}\ \frac{s_s|e \models EG\varphi}{s_s|e \models \varphi \wedge EXEG\varphi} \qquad \texttt{unr}_{ag}\ \frac{s_s|e \models AG\varphi}{s_s|e \models \varphi \wedge AXAG\varphi}$$

$$\texttt{unr}_{s_s,ex}\ \frac{s_s|e \models EX\varphi}{s_i|e_i \models \varphi,\ s_j|e_j \models tt \ldots s_k|e_k \models tt} \begin{cases} s_i \in NS = \{s_I | s \xrightarrow{I}_{\mathcal{M}} s_I\}, s_{j,\ldots,k} \in NS - \{s_i\} \\ e_{i,j,\ldots,k} \in NS_e = \{e_I | e \xrightarrow{I}_{\mathcal{E}} e_I\} \end{cases}$$

$$\texttt{unr}_{s_s,ax}\ \frac{s_s|e \models AX\varphi}{s_1|e_1 \models \varphi\ \ldots\ s_k|e_k \models \varphi}\ \forall 1 \leq i \leq k . s_s \xrightarrow{i}_{\mathcal{M}} s_i \wedge e \xrightarrow{i}_{\mathcal{E}} e_i$$

Fig. 3. Tableau Rules for Module Checking System States

φ; i.e.

$$\mathcal{M} \models_o \varphi \Leftrightarrow \forall \mathcal{E}.\mathcal{M} \times \mathcal{E} \models \varphi \tag{3}$$

Proceeding further, from Equation 3 we state that

$$\mathcal{M} \not\models_o \varphi \ \Leftrightarrow\ \exists \mathcal{E}'.(\mathcal{M} \times \mathcal{E}' \not\models \varphi \Leftrightarrow \mathcal{M} \times \mathcal{E}' \models \neg\varphi) \tag{4}$$

It is important to note however that $\mathcal{M} \not\models_o \varphi \not\Rightarrow \mathcal{M} \models_o \neg\varphi$. It can be shown that a module does not satisfy both a formula and its negation [4]. A module might satisfy a formula and its negation under different environments. For example: given $\mathcal{M} = (\{s^0, s_1, s_2\}, s^0, \{s^0 \xrightarrow{1} s_1, s^0 \xrightarrow{2} s_2, s_1 \xrightarrow{1} s_1, s_2 \xrightarrow{1} s_2\}, \{p\}, L, 2\})$ where $L(s_1) = \{p\}$ and CTL formula AXp, it is easy to see that $\mathcal{M} \not\models_o AXp$ and $\mathcal{M} \not\models_o EX\neg p$.

3.1 Local Module Checking and Generation of Witness

We present here a tableau-based technique similar to [4] for constructing the witness environment $\mathcal{E}$, existence of which ensures that the module does not satisfy original formula. Tableau rules are defined as

$$\frac{\texttt{Antecedent}}{\texttt{Consequent}}$$

where the `Antecedent` represents the current obligation for module checking and `Consequent` denotes the next obligation. A successful tableau (see below) will result in automatic generation of the environment $\mathcal{E}$. Figs. 3 and 4 present the complete tableau where the former corresponds to the rules for system states and the latter for the environment-controlled states.

[4] Note that the same is not true for model checking problem: $M \not\models \varphi \Leftrightarrow M \models \neg\varphi$

$$\texttt{reorg}\ \frac{s_e|e \models \varphi}{s_e|e \models \{\varphi\}}$$

$$\texttt{emp}\ \frac{s_e|e \models \{\}}{\cdot} \qquad \texttt{prop}\ \frac{s_e|e \models \{p,C\}}{s_e|e \models C}\ p \in L(s_e)$$

$$\wedge\ \frac{s_e|e \models \{\varphi_1 \wedge \varphi_2, C\}}{s_e|e \models \{\varphi_1, \varphi_2, C\}} \qquad \vee_1\ \frac{s_e|e \models \{\varphi_1 \vee \varphi_2, C\}}{s_e|e \models \{\varphi_1, C\}} \qquad \vee_1\ \frac{s_e|e \models \{\varphi_1 \vee \varphi_2, C\}}{s_e|e \models \{\varphi_2, C\}}$$

$$\texttt{unr}_{eu}\ \frac{s_e|e \models \{E(\varphi U \psi), C\}}{s_e|e \models \{(\psi \vee (\varphi \wedge EXE(\varphi U \psi))), C\}} \qquad \texttt{unr}_{au}\ \frac{s_e|e \models \{A(\varphi U \psi), C\}}{s_e|e \models \{(\psi \vee (\varphi \wedge AXA(\varphi U \psi))), C\}}$$

$$\texttt{unr}_{eg}\ \frac{s_e|e \models \{EG\varphi, C\}}{s_e|e \models \{\varphi \wedge EXEG\varphi, C\}} \qquad \texttt{unr}_{ag}\ \frac{s_e|e \models \{AG\varphi, C\}}{s_e|e \models \{\varphi \wedge AXAG\varphi, C\}}$$

$$\texttt{unr}_{s_e}\ \frac{s_e|e \models C}{\exists \pi \subseteq \Pi.\ \exists \Pi_{C_{ex}}(\pi).\ \forall i \in \pi.\ s_i|e_i \models C_{ax} \cup C_i}$$

$$\begin{cases} C_{ax} = \bigcup_{AX\varphi_k \in C} \varphi_k \\ C_{ex} = \bigcup_{EX\varphi_l \in C} \varphi_l \\ \Pi = \{i \mid s_e \xrightarrow{i}_{\mathcal{M}} s_i\} \\ \Pi_{C_{ex}}(\pi) = \{C_i \mid i \in \pi \subseteq \Pi \wedge C_i \subseteq C_{ex}\} \\ C_{ex} = \bigcup_{i:C_i \in \Pi_{C_{ex}}(\pi)} C_i \\ \bigcap_{i:C_i \in \Pi_{C_{ex}}(\pi)} C_i = \Phi \end{cases}$$

Fig. 4. Tableau Rules for Module Checking Environment States

Tableau for System States. Consider first the Fig. 3 (without the `reorg` rule). The rules for `prop`, $\wedge$, $\vee$s are simple and intuitive. The `prop` rule leads to a successful tableau leaf, the $\wedge$ rule is successful if both its consequents are successful and finally, the success of $\vee$-rule depends on the success of any of its consequents. The rule $\texttt{unr}_{eu}$ corresponds to *unrolling* of the EU formula expression. A state satisfies $E(\varphi U \psi)$ iff (a) ψ is satisfied in the current state or (b) φ is true in the current state and in one of its next states $E(\varphi U \psi)$ is satisfied. The rule for $\texttt{unr}_{au}$ is similar to $\texttt{unr}_{eu}$ with exception of the presence of universal quantification on the next states (AX). The rule $\texttt{unr}_{eg}$ ($\texttt{unr}_{ag}$) states that the current state satisfies φ and in some (all) next state $EG\varphi$ ($AG\varphi$) holds true.

Finally, the rules for $\texttt{unr}_{s_s,ex}$ and $\texttt{unr}_{s_s,ax}$ correspond to the unfolding of the state and the formula expression simultaneously. Note that in the former, we are searching for at least one next state, while in the latter all next states should satisfy φ. As such for the EX-formula expression, the tableau selects any one of the next states $s_i|e_i$ and if the selected state satisfies φ, there is no obligation left for the rest of the next states; the obligations on the remaining next states $s_{j,\ldots,k}$ in the context of the environment is to satisfy tt [5] (any state satisfies the propositional constant tt). Note that, the tableau can potentially have k sub-tableaus for $\texttt{unr}_{s_s,ex}$ each of which will correspond to selection of one next state s_i from the set of k next states

[5] For the purpose of constructing the environment, in Rule $\texttt{unr}_{s_s,ex}$, we can safely assume that all the environment states $e_{j\ldots,k}$ replicates the behavioral patterns of module states $s_{j\ldots,k}$.

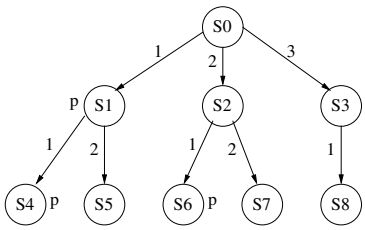
of s_s.

Observe that, the rules $\mathtt{unr}_{s_s,ex}, \mathtt{unr}_{s_s,ax}$ lead to *one-step* construction of the environment. Conforming to the constraint that the environment at the system state cannot control the enabled transition, the environment must have exactly the same number of transitions as the system state.

Tableau for Environment-Controlled States. The tableau rules (Fig. 4) for environment controlled states are slightly different from the one described above. Instead of asking whether a state satisfies a formula expression (see Fig. 3), the question asked is whether a state satisfies a set of formula expressions. In fact the set represents a formula expression equivalent to conjunction over its elements. The reason for altering the tableau rule structure stems from the fact the environment plays an active role in deciding the enabled transitions. For example: in order to construct an environment state e such that $s_e|e \models \varphi \wedge \psi$, we need to construct e_1 and e_2 such that $s_e|e_1 \models \varphi$ and $s_e|e_2 \models \psi$ with the constraint that $e_1 = e_2$, i.e. exactly the same set of transitions is enabled to module check φ and ψ at the state s_e. To address to this state of affairs, *the tableau rules for environment-controlled state maintains a global view of all the formula to be satisfied and ensures consistent enabling/disabling of transitions by the environment (to be constructed).*

The rules for $\mathtt{prop}, \vee, \mathtt{unr}_{eu}, \mathtt{unr}_{au}, \mathtt{unr}_{eg}, \mathtt{unr}_{ag}$ in Fig. 4 are similar to that in Fig. 3. The rule for $\wedge$ aggregates all the conjuncts in the set. The $\mathtt{emp}$-rule represents the case the state does not have obligation to satisfy any formula and a (successful) tableau leaf is reached. The rule $\mathtt{unr}_{s_e}$ is applied only when no other rules are applicable. In other words, the set C only contains EX and/or AX formula expressions. C_{ax} is the set of all formula expressions that must be satisfied in all next states while C_{ex} is the set of the formula expressions each of which must be satisfied in at least one of the next states. Π records all the indices of the outgoing transitions from s_e, while $\Pi_{C_{ex}}(\pi)$ is a subset of C_{ex} such that there is at least one subset for each i present in a subset $\pi \in \Pi$. For example, if π is a singleton set, then $\Pi_{C_{ex}}(\pi)$ is also a singleton set containing C_{ex}. In short, $\Pi_{C_{ex}}(\pi)$ is used to associate with i-th selection of next state-environment pair a set of elements $C_i \subseteq C_{ex}$. The consequent of the rule, therefore, fires the obligation that all states identified by the indices in π must satisfy C_{ax} and the corresponding subset of C_{ex} as identified by $\Pi_{C_{ex}}(\pi)$.

This rule is illustrated by Fig. 5. All states in the Fig. 5(a) are environment states and the proposition p is true at states s_1, s_4, and s_6. The obligation at $s_0|e_0$ is to satisfy $C = \{AXEXp, EXEX\neg p, EXp\}$. As there are 3 transitions from s_0 there are $2^3 - 1 = 7$ different choices for π. Fig. 5(b) shows subsets consisting of only 1 and 2. π represents the indices of enabled transitions. These transitions lead to states which must satisfy all the elements of $C_{ax} = \{EXp\}$. Corresponding to each π, there exists at least one choice for $\Pi_{C_{ex}}(\pi)$ which subsets $C_{ex} = \{p, EX\neg p\}$ in $|\pi|$ subsets where $|\pi|$ is the size of π. It also assigns each subset to different subset of next states where elements of the assigned subset must be satisfied. For example for $\pi = \{1, 2\}$, there are two possible ways of assigning subsets of C_{ex} to $s_1|e_1$ and $s_2|e_2$ (see Fig. 5(b)). In this example, we obtain an environment e_0 for $\pi = \{1, 2\}$ (i.e.



(a)

Antecedent: $s_0|e_0 \models C \quad C_{ax} = \{EXp\}, C_{ex} = \{p, EX\neg p\}$

π	$\Pi_{C_{ex}}(\pi)$	Consequent $s_i\|e_i \models C_{ax} \cup C_i$
{1}	$\{C_{ex}\}$	$s_1\|e_1 \models \{EXp, EX\neg p, p\}$
{2}	$\{C_{ex}\}$	$s_1\|e_1 \models \{EXp, EX\neg p, p\}$
{1,2}	$\{\{p\},\{EX\neg p\}\}$ ✓	$s_1\|e_1 \models \{EXp, p\}$ $s_2\|e_2 \models \{EXp, EX\neg p\}$
	$\{\{EX\neg p\},\{p\}\}$	$s_1\|e_1 \models \{EXp, EX\neg p\}$ $s_2\|e_2 \models \{EXp, p\}$
⋮		

(b)

Fig. 5. Illustration of unr_{s_e} tableau rule

the transition labeled 3 from s_0 is disabled), and $\Pi_{C_{ex}}(\pi) = \{\{p\}, \{EX\neg p\}\}$. Note that, our local approach does not necessarily examine all possible choices for π and the corresponding subsets $\Pi_{C_{ex}}(\pi)$; instead it terminates as soon as an environment that leads to satisfiability of given formula is obtained.

Finally, consider the `reorg` (re-organize) rules in Figures 3 and 4. These rules rearrang the formula obligations from set-based to expression-based or vice versa depending on the type of the model state being considered.

Finitizing the Tableau. The given tableau rules can be of infinite depth as each recursive formula expressions AU, EU, AG, EG are unfolded infinitely many times. However, the total number of states in the Kripke structure for the module is $N = |S|$, and this finitizes the tableau depth. In Fig. 3, if the pair (s_s, φ) in $s_s|e \models \varphi$, where φ is either EG or AG formula expression, appears twice in a tableau path, we fold back the tableau by pointing the second occurrence to the first and stating a successful tableau loop is obtained. Note that this also leads to generation of a loop in the constructed environment. On the other hand, if the pair (s_s, φ), where φ is of the form EU or AU, appears twice, the second occurrence is classified as an unsuccessful tableau leaf and is replaced by *ff*.

The idea of folding back or replacing using false relies on fixed point semantics of CTL formulas. The CTL formulas EG and AG can be represented by greatest fixed point recursive equations:

$$EG\varphi \equiv Z =_\nu \varphi \wedge EXZ \quad AG\varphi \equiv Z =_\nu \varphi \wedge AXZ$$

In the above ν represents the sign of the equation and is used to denote greatest fixed point and Z is recursive variable whose valuation/semantics (set of model states) is the greatest fixed point computation of its definition. Similarly the fixed point representation of CTL formulas AU and EU are

$$E(\varphi U\psi) \equiv Z =_\mu \psi \vee (\varphi \wedge EXZ) \quad A(\varphi U\psi) \equiv Z =_\mu \psi \vee (\varphi \wedge AXZ)$$

The fixed point computation proceeds by iteratively computing the approximations of Z over the lattice of set of states in the model. A solution is reached only when two consecutive approximations are identical. For greatest fixed point computation, the first approximation is the set of the all states (top of the lattice) and

as such a system can satisfy a greatest fixed point formula along an infinite path (using loops). On the other hand, the first approximation of the least fixed point variable is empty set (bottom of the lattice) and therefore satisfiable paths for least fixed point formula are always of finite length.

For the tableau in Fig. 4, the finitization condition is similar. If the pair (s_e, C) in $s_e|e \models C$, appears twice in a tableau path and if C contains any least fixed point CTL formula expression, then the second occurrence is replaced by ff; otherwise the second occurrence is made to point to the first and a successful tableau is obtained.

Theorem 3.3 (Sound and Complete) *Given a module $\mathcal{M}$ and CTL formula φ, $\mathcal{M} \not\models_o \varphi$, iff the tableau in Figures 3 and 4 generates an environment $\mathcal{E}$ where $\mathcal{M} \times \mathcal{E} \models \neg\varphi$.*

Proof. The proof proceeds by realizing the soundness and completeness of each of the tableau rules. For brevity, we present here the proof-sketch for $\mathtt{unr}_{s_e}$, proofs for the other rules are straightforward.

Recall that, $s_e|e \models C$, where C is the set of formula expressions with temporal operators AX and EX, is satisfiable if the next states proof obligations are satisfied by destination states reachable via transitions enabled by the environment e. The environment can enable any subset (barring $\emptyset$) of transitions. The tableau rule, therefore, considers all possible subset of destination states of enabled transitions.

As each transition is annotated by an index (whose domain is over the out-going branching factor of s_e), we construct Π, the set of indices of out-going transitions. In other words, $i \in \Pi \Rightarrow s_i$ is reachable via the transition with index i. We are required to identify one possible subset of Π which represents the enabled transitions whose destinations conform to the satisfiability obligations in the consequent (see $\exists\pi \subseteq \Pi$ in the consequent). Let $\pi_s = \{s_i \mid i \in \pi\}$ be the next states reachable via (selected) enabled transitions.

The consequent of the tableau rule has the following obligations. All elements of π_s in parallel composition with the environment must satisfy the expressions in C_{ax} and for each formula expression φ in C_{ex}, there must be at least one state which in conjunction with the corresponding environment satisfies φ. Observe that, there is requirement for an existence of a subset, $\Pi_{C_{ex}}(\pi)$, of C_{ex} corresponding to a subset π such that next state-environment pairs satisfy the corresponding obligations. This ensures that the environment constructed is consistent, i.e., e_i is constructed such that $s_i|e_i$ satisfies both the for all obligations (C_{ax}) and its share of existential obligations (C_i). Therefore, if we can generate an environment for s_e corresponding to rule $\mathtt{unr}_{s_e}$, then $s_e \not\models_o \psi$ where ψ is the disjunction of the elements of the set $\{AX\neg\varphi \mid \varphi \in C_{ex}\} \cup \{EX\neg\psi \mid \psi \in C_{ax}\}$. The other direction can be proved likewise. □

Complexity.

The main factor in complexity is attributed to the handling of universal and existential formulas in $\mathtt{unr}_{s_e}$ rule in Fig. 4. The number of different obligations that can be fired on the basis of each selection of $\pi \subseteq \Pi$, is $O(2^{|C_{ex}|\times|\pi|})$ where $|C_{ex}|$

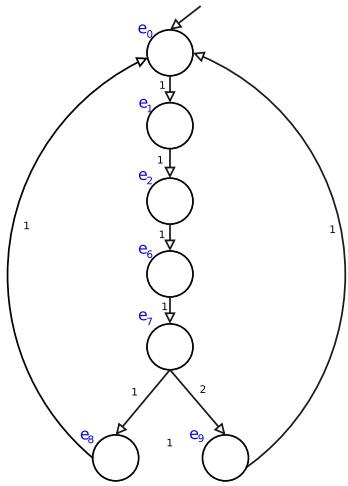


Fig. 6. The witness environment for the coffee brewer example

and $|\pi|$ are respectively the size of the respective sets. This is because we need to consider all possible permutations of elements of C_{ex} and match the permutations with all possible permutations for selecting element from π. Overall complexity is, therefore, exponential to the maximum branching factor (maximum size of π) of the module times the size of the formula to be satisfied. It is worth noting here that if the given formula is free from EX and EU, the complexity of tableau-based approach will be polynomial to the sizes of the formula and module (same as model checking). This is due to the fact that in the presence of only AX-formulas in rule $\mathtt{unr}_{s_e}$, we are only required to find one element from Π; the state corresponding to which satisfies C_{ax}.

Figure 6 shows the witness environment generated for the coffee brewer in Figure 1. The witness environment disables three transitions of state s_1 such that there is no path where ten cups can be selected. Under this witness, it can be shown that the model does not satisfy the original property $AG\,E(tt\,U\,TEN \wedge AF(SERVE \vee ERROR))$.

4 Implementation

A local module checking tool has been implemented using C/C++ and many SMV benchmarks from the NuSMV package have been tested. The tool proceeds as follows:

(i) An SMV model is converted into an explicit state FSM using NuSMV. This is achieved by traversing the model's state space in NuSMV and writing all reachability information (states, transitions and labels) to a file. As there is no explicit notion of system/environment states in NuSMV, the state space is divided randomly into two sets of equal size representing system and environment states respectively.

(ii) The file containing the explicit state FSM is read by the tool along with the

CTL property to be used for verification.

(iii) The CTL property is negated and the negation is carried inwards. CTL properties can not have negations applied to formulas other than propositions.

(iv) A search for a witness is carried out, starting with the initial state, and the tool attempts to generate an environment under which the module satisfies the negated CTL property. If a witness is present, then the module does not satisfy the original property.

The algorithm applies the appropriate system or environment state tableau rules on the current state. Once all present (current-state) commitments are met, any future commitments are passed to its successors. It uses a heuristic to compute a small set of successors of the current state which satisfy all its future commitments. This is used to ensure that the generated witness is small. If no witness can be computed, it can be concluded that the module satisfies the CTL property. Note that the algorithm attempts to generate a small witness and not the minimal witness for a given property and module. In order to compare the obtained results, the tool was extended to find an environment under which the original CTL property is satisfied. This is achieved as follows:

(i) The CTL property (non-negated) is read along with the module FSM (as above).

(ii) The algorithm attempts to find an environment under which the given CTL property is satisfied by traversing all of the reachable state space.

It is important to note here that the above is *not* an implementation of global module checking. Global module checking advocates the need to check if the given property is satisfied by the module under all environments. However, the above approach constructs only a single environment under which the given CTL property is satisfied by the module. However, unlike the local module checker which attempts to generate a small witness, the algorithm for global module checking constructs the biggest environment under which the module satisfies the given property. This is done by enabling all but those transitions in reachable environment states which may lead to the dissatisfaction of the given property. It was observed that on the average, computing the maximal environment under which the original CTL property is satisfied consumes more time than computing a small witness under which it fails.

The problem of computing all possible environments (global module checking) is even more harder and time consuming and the benefits of local module checking will be even more apparent in such a case.

5 Experimental Results

The results are given in table 1. The first column contains the name and size (in number of states) of the verified module and the CTL property used is given in the second column. The results obtained from local module checking are presented in

System($\|S\|$)	CTL Property	Local module checking	Generation of maximal witness
short(4)	$AG(request- > AFstate = busy)$	0.02/F/0 (2)	0.05/S/0
ring(7)	$(AGAFgate1.output) \wedge (AGAF!gate1.output)$	0.01/F/0 (2)	0.02/S/0
Counter(8)	$AG(bit2.value = 0)$	0.00/S/0 (5)	0/F/0
coffee(10)	$AGAF(ten \wedge EFServe)$	0.0084/S/3 (7)	0.001/S/0
MCP(30)	$AGEF((MCP.missionaries = 0) \wedge (MCP.cannibals = 0))$	0.001/S/5 (2)	0.05/S/0
base(1758)	$trueAU(step = 0)$	1.250/F/0 (21)	1.290/S/0
periodic(3952)	$AG(aux \neq p11)$	9.580/S/0 (701)	51.270/F/0
syncarb5(5120)	$AGEFe5.Persistent$	7.73/S/0 (223)	3704/S/960
dme1(6654)	$AG((e-1.r.out = 1 \mid e-2.r.out = 1) \wedge(e-1.r.out = 1 \mid e-3.r.out = 1) \wedge(e-2.r.out = 1 \mid e-3.r.out = 1))$	5.490/F/0 (141)	41.17/S/790
pqueue(10000)	$EG(out\lrcorner[1] = 0)$	34.20/F/0 (1904)	35.130/S/0
pqueue(10000)	$AF(out\lrcorner[1] = 0)$	34.930/F/0 (2101)	34.960/S/0
barrel(45025)	$AGtrueAU(b0 = 0)$	12.720/S/0 (231)	34.190/S/1
idle(87415)	$trueAU(step = 0)$	77.190/F/0 (38088)	79.920/S/0
abp4(275753)	$EF(sender.state = get)$	130.77/F/0 (59808)	133.880/S/0

Table 1
Implementation Results

the third column. The fourth column contains the results of generating the maximal environment under which the given CTL property is satisfied [6].

Note that for many modules, the original property and its negation were both satisfied under different environments. A majority of models had multiple start states. In these cases, the local module checker (and the maximal witness generater) was executed on each start state. Models with dense transition relations such as dme1 and syncarb5 took significantly more time. Models with relatively sparse transitions relations such as abp4 and idle took lesser time even though they had a higher number of reachable states. The local module checker took slightly longer when the original CTL formula was satisfied by the module (abp4, pqueue, periodic).

6 Conclusions

Module checking extends model checking for open systems. It has been shown in [10] that the complexity of module checking for branching time logic CTL is EXPTIME complete. The above approach to module checking generates all possible environments so that the composition of the model and the environment satisfies the CTL property. In this paper we propose a local approach to CTL module checking. The proposed approach tries to determine a witness environment so that the negation of the property is satisfied by the composition of the witness and the model. When this is possible, the original property is not satisfied over the module. We have developed a set of sound and complete tableau rules for local

[6] The results are in format TimeTaken(seconds)/Result(SUCCESS or FAILURE)/Number of disablings (Number of states traversed locally during local module checking).

module checking of CTL. The efficiency of the proposed approach is demonstrated by comparing the performance of local and traditional global module checking using benchmarks from NuSMV. The results presented compare the generation of one environment in both cases. Answering the module checking question using a global strategy requires the generation of all environments, which will be computationally much more expensive than the local approach.

References

[1] Girish Bhat, Rance Cleaveland, and Orna Grumberg. Efficient on-the-fly model checking for CTL*. In *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 388–397, June 1995.

[2] R. Cavada, Alessandro Cimatt, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV 2.1 User Manual*, June 2003.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–748, 1990.

[5] Matthew B. Dwyer and Corina S. Pasareanu. Filter-based model checking of partial systems. In *Foundations of Software Engineering*, pages 189–202, 1998.

[6] Patrice Godefroid. Reasoning about abstract open systems with generalized module checking. In *EMSOFT'2003 (3rd Conference on Embedded Software)*, pages 223–240, 2003.

[7] D. Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[8] O. Kupferman and MY. Vardi. Module checking [model checking of open systems]. In *Computer Aided Verification. 8th International Conference, CAV '96*, pages 75–86, Berlin, Germany, 1996. Springer-Verlag.

[9] Orna Kupferman and Moshe Y. Vardi. Module Checking Revisited. In *CAV*, 1997.

[10] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. Module checking. *Information and Computation*, 164:322–344, 2001.

[11] Z. Manna and A. Pnueli. A temporal proof methodology for reactive systems. In *Program Design calculi, NATO ASI Series, Series F: Computer and System sciences*. Springer-Verlag, 1993.

[12] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[13] A. Pnueli. The temporal logic of programs. In *18th IEEE Symp. found. Comp. Sci. (FOCS)*, 1977.

[14] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 1981.

[15] P. S. Roop and A. Sowmya. Forced simulation: A technique for automating component reuse in embedded systems. In *ACM Transactions on Design Automation of Electronic Systems*, October 2001.

[16] M. Y. Vardi. Verification of concurrent programs: The automata theoretic framework. In *2nd IEEE Symp. on Logic in Computer Science*, 1987.

Appendix

Example. Table 2 describes the steps involved in generating a witness environment for the coffee brewer example in Figure 1. First, the original CTL property $AG\,E\,(ttUTEN \wedge AF(SERVE \vee ERROR))$ is negated (and the negation carried inwards) to $E(ttU\,AG\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR))$. The local module checker is then called on the initial state (s^0) of the model along with the negated

$s^0|e_0 \models \{E(tt\ U\ AG\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR))\}$

$s^0|e_0 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)$
$\vee\ (tt\ \wedge\ EXE(tt\ U\ AG\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)))\}$

$s^0|e_0 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

$s^0|e_0 \models \{\neg TEN\ ,\ AXAG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

$s_1|e_1 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

$s_1|e_1 \models \{\neg TEN\ ,\ AXAG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

↙

$s_2|e_2$ OR $s_3|e_3$ OR $s_4|e_4$ OR $s_5|e_5 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}\}$

$s_2|e_2 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}\}$ (s_3, s_4, s_5 not explored)

$s_2|e_2 \models \{\neg TEN\ ,\ AXAG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

↙

$s_6|e_6 \models AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)$

$s_6|e_6 \models \neg TEN \wedge AXAG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)$

↙

$s_7|e_7 \models AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)$

$s_7|e_7 \models \neg TEN \wedge AXAG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)$

↙

$s_8|e_8\ AND\ s_9|e_9 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}\}$

$s_8|e_9\ AND\ s_9|e_9 \models \{\neg TEN\ ,\ AXAG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

↙

$s^0|e_0 \models \{AG(\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR)\}$

Table 2
Tableau for constructing witness environment for the coffee brewer example in Figure 1

formula. The algorithm applies the appropriate tableau rules described earlier and first attempts to check if the current state satisfies all current-state commitments. Then the next-state commitments are passed on to the successors.

For example, initially the negated property $E(ttU\ AG\neg TEN \vee AG(\neg SERVE \wedge \neg ERROR))$ is passed to the initial state s^0 of the model. The negated property is then broken down (using the environment state tableau rule $\mathtt{unr}_{eu}$) to $\{AG\,(\neg TEN\ \vee\ AG(\neg SERVE\ \wedge\ \neg ERROR)\ \vee\ \ (tt\ \ \wedge\ \ EX\,EttU\,AG(\neg TEN\ \vee\ AG(\neg SERVE\ \wedge\ \neg\,ERROR)))$. The resulting disjunction is then broken further (using the tableau rule $\wedge$). Once the current-state commitments are met, all next state commitments of s^0 are passed to its successor s_1. This is shown using ↙ in table 2. Note that for environment states, formulas are organized into set notation whereas for system states, they are applied in the order they arrive. The algorithm terminates when a strongly connected component which satisfies the negated property is found.