# A One-Pass Tableau-Based Workflow Verification Framework

Md Zahidul Islam and Wendy MacCaull

Centre for Logic and Information
St. Francis Xavier University, Antigonish, Canada
{x2010mce, wmaccaul}@stfx.ca

**Abstract**

Workflow management systems (WfMSs) are useful tools for supporting enterprise information systems. Such systems must ensure compliance with guidelines and regulations. While formal verification techniques can be used in the development stages to help ensure behavioral properties of many systems, these techniques are generally not available in workflow tools. We present a framework which models workflows using Petri nets and translates the model to a tableau style model checker. The model checker uses the recently introduced one-pass tableau algorithm and delivers enhanced performance over traditional two-pass strategies in practical applications. A failed tableau will generate a counter model which can aid in debugging. We present a case study involving a health services delivery program, and verify properties written in Computation Tree Logic (CTL). The tableau method can be modified to accommodate other specification languages such as timed CTL, logics of beliefs, desires and intentions, temporal description logic, first order logic, and others.

## 1 Introduction

It is a common practice to analyze a system's behaviour before its actual implementation. Established analysis approaches like test beds allow for rigorous, transparent, and replicable testing of software, hardware, and networking systems. However, they are difficult to set up, are usually very costly, and require experts. Another approach, simulation, involves providing certain inputs and observing their corresponding outputs. Providing all possible inputs and observing their outputs is tedious and usually impractical. These shortcomings led researchers to the application of formal verification in system analysis and development. This involves modelling systems using an adequate level of abstraction which decreases analysis cost and gives a rigorous view of the system. Models are relatively easy to modify and errors found before implementation can greatly decrease cost. Properly verified models ensure better processes. We present a framework for applying formal verification to workflow models.

There are two formal verification approaches: theorem proving and model checking. Theorem proving is a logic based proof theoretic approach which typically uses a very expressive language for describing systems and property specifications. The system is expressed as a set of axioms and the specifications are expressed as formulae; a proof system is used to determine if the formulae are valid. In model checking, the description of a system (also known as a model) is given by the specification language of a model checker and the model checker determines if a (usually) temporal logic (such as Computation Tree Logic (CTL), or Linear Temporal Logic (LTL)) formula holds for the model. Applying a formal verification technique by modelling a system using a formal specification language is generally a difficult and tedious task. Our framework starts with a human-friendly specification language, Petri nets, which has a graphical representation easily modelled with the Coloured Petri net (CPN) tool [23]; the Petri net models are automatically transformed to Kripke structures for formal analysis.

The system does not have to be represented in the formal specification language of an existing model checker.

The tableau method is a popular proof procedure applicable to a wide range of logics including temporal logics. The traditional tableau method for CTL is a two-pass procedure [2] which applies a set of tableau rules to construct a tableau in the first pass and determines the inconsistent nodes (the nodes that cannot be a part of a valid model for the given CTL formula) in the second pass. A formula is satisfiable if and only if a model for the formula can be found in the tableau. Recently an improvement over the two-pass procedure, known as the one-pass tableau procedure [1], was developed. A comparison between the one-pass and two-pass procedure in [8] showed that the one-pass procedure consistently, and sometimes dramatically, outperformed the two-pass procedure. The one-pass procedure requires a single pass through the tree to determine the satisfiability of a formula. This procedure can be used for model checking where the tableau is constructed from a given CTL property and a system model. Here we propose a workflow verification framework based on this technique. Our framework includes an automatic translator to translate a Petri net workflow model to the corresponding Kripke structure, and uses the one-pass tableau procedure for model checking. One-pass tableau-based decision procedures have been used for various logics, but to the best of our knowledge we are the first to use the one-pass tableau algorithm for model checking. We show the usefulness of our framework with a case study involving health service delivery.

The rest of this paper is organized as follows: Section 2 presents some related work; Section 3 presents some background topics; Section 4 discusses various components of the one-pass tableau-based workflow verification framework; Section 5 describes a case study, and Section 6 concludes the paper and offers some directions for future work.

## 2 Motivation and Related Work

There is no foundational, well recognized, or universally accepted formalism for workflow verification [7]. In [19], the authors discussed an automatic translation of workflow models into DVE, the specification language of the DiVinE model checker. The end result of their work is the NOVA WorkFlow [6] tool which uses the Compensable Workflow Modelling Language (CWML) for workflow modelling. Use of the DiVinE model checker limited the NOVA WorkFlow tool to LTL property specifications. An application of the SPIN model checker to workflow verification can be found in [22], and another approach using UPPAAL is available in [10]. Spin supports LTL, and UPPAAL supports Timed Computation Tree Logic. A tableau-based model checker for Temporal Description Logic (ALCT) can be found in [4] and a similar approach for Timed $\text{BDI}_{CTL}$ can be found in [15]. Both the ALCT and Timed $\text{BDI}_{CTL}$ model checkers use Petri nets to design workflows, but the Petri net models were translated manually to generate the state space for model checking.

Among the other workflow management systems, FlowMake [20] can identify structural conflicts in process models, YAWL [24] has some verification facilities mainly with respect to structural properties, AgentWork [16] uses dynamic rules to allow users to identify errors in the execution logic of the workflow while $\text{PLM}_{flow}$ [26] can generate workflow from business rules and can detect deadlocks.

Existing tools involve the usually difficult task of writing workflow models in the specification language of existing model checkers, or lack temporal verification capabilities, or are restricted to one property specification language. Tools and techniques to support workflow modelling and automatic verification in a single but flexible framework are needed.

According to [9], with suitable optimization techniques, tableau-based methods are potentially more flexible and efficient than other model checking approaches. The one-pass tableau strategy has been developed recently [1], and a naïve implementation of the one-pass tableau-based decision procedure for various logics [1] is available at [17]. While the worst-case complexity of the one-pass algorithm is 2EXPTIME which is worse than the EXPTIME complexity of the two-pass algorithm, in most practical situations, the worst case rarely arises; indeed the one-pass algorithm consistently outperforms the two-pass algorithm [8].

# 3 Preliminaries

In our framework, we use Petri nets to formally describe a workflow model, CTL to describe properties of the system, and the one-pass tableau-based model checking algorithm.

## 3.1 Workflow modelling using Petri nets

Worflow management systems (WfMSs) such as YAWL [24] provide users without any programming experience a relatively easy way to organize and/or describe complex processes in a visual format. Financial institutions, healthcare organizations, etc., involving complex processes, information and communication systems are adopting WfMSs to orchestrate the various activities. The main objective of workflow modelling is to provide an abstract view of a system to support analysis. Petri nets provide a graphical interface along with a strong mathematical foundation to accomplish this. A Petri net is a graph with two types of nodes—*places* and *transitions*. Arcs connect the two types, and no two nodes of the same type can be directly connected.

**Definition 1** (Petri net)**.** *A Petri net is a triple* $(P, T, F)$*, where:* $P$ *is a finite set of places,* $T$ *is a finite set of transitions* $(P \cap T = \emptyset)$*, and* $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation).*

Researchers developed various workflow patterns to facilitate the development of process-oriented applications. For this paper, we consider only the basic control flow patterns, namely sequential flow, parallel flow, conditional flow, and iterative flow [14]. These suffice to define many complex workflows. Our framework can handle Petri nets workflows containing all the four basic control flow patterns. Among the four basic patterns, conditional flow and iterative flow require special care. For example, in conditional flow, the token will follow one of the paths; in the iterative flow, the token will follow the same path multiple times. We used a variable called *guard* to restrict the token passing along one of the paths or along the same path. We also need to determine when to terminate an iterative flow which require updating a variable with each iteration. We call this updating step an *action*.

In Petri nets, the state space is given implicitly, but for formal verification, we need to generate the explicit states or markings. A marking $M$ of a Petri net is a function from the set of places $P$ to the non negative integers. Firing a transition $t$ in a Petri net with marking $M$, results in a new marking $M'$. A Petri net can be represented as a Kripke structure: the states are markings and there is a transition in the Kripke structure from $M$ to $M'$ whenever a transition in the Petri net creates a marking $M'$ from $M$.

[1] PDL, CTL, LTL, Modal Logic KD, KD45, K4, CK, K, S4, KLM Logic P, Intuitionistic Logic G4IP, and Propositional Classical Logic

**Definition 2** (Kripke Structure)**.** *A Kripke structure, over a set AP of atomic propositions, is a 4-tuple* $\mathcal{M} = \big(S, \rightarrow, L, I\big)$*, where S is a finite set of states,* $\rightarrow \subseteq S \times S$ *is a transition relation,* $L : S \rightarrow 2^{AP}$ *is an interpretation function, and* $I \subseteq S$ *is a set of initial states.* $L(s)$ *is the set of atomic propositions satisfied by s.*

## 3.2 Property specifications

We chose CTL as the property specification language. The syntax and semantics of CTL are available in [1]. In addition to propositional operators, CTL has path quantifiers, $A$ (all paths), $E$ (some paths), and temporal modalities, $G$ (all future states), $F$ (some future state), $X$ (the next state), $U$ (until) and $B$ (before). In a formula, a temporal operator must be preceded by a path quantifier. The inductive definition of CTL formulae in Backus Näur Form is given below:

$$\begin{aligned}\varphi \ ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid EX\ \varphi \mid\ AX\ \varphi \mid EF\ \varphi \mid AF\ \varphi \mid \\ EG\ \varphi \mid AG\ \varphi \mid E[\varphi U \varphi] \mid A[\varphi U \varphi] \mid E[\varphi B \varphi] \mid A[\varphi B \varphi]\end{aligned}$$

where $p$ ranges over a set of atomic propositions. Given a Kripke structure $\mathcal{M}$ and a CTL formula $\varphi$, the semantics of CTL is defined as follows:

1. $\mathcal{M}, s \models \top$ and $\mathcal{M}, s \not\models \bot$.
2. $\mathcal{M}, s \models p$ if and only if $p \in L(s)$.
3. $\mathcal{M}, s \models \neg\varphi$ if and only if $\mathcal{M}, s \not\models \varphi$.
4. $\mathcal{M}, s \models \varphi_1 \wedge \varphi_2$ if and only if $\mathcal{M}, s \models \varphi_1$ and $\mathcal{M}, s \models \varphi_2$.
5. $\mathcal{M}, s \models \varphi_1 \vee \varphi_2$ if and only if $\mathcal{M}, s \models \varphi_1$ or $\mathcal{M}, s \models \varphi_2$.
6. $\mathcal{M}, s \models EX\ \varphi$ if and only if $\exists s' \in S,$ such that $s \rightarrow s'$ and $\mathcal{M}, s \models s'$.
7. $\mathcal{M}, s \models AX\ \varphi$ if and only if $\forall s' \in S,$ if $s \rightarrow s'$ then $\mathcal{M}, s \models s'$.
8. $\mathcal{M}, s \models EG\ \varphi$ if and only if there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$, and for all $s_i$ along the path, we have $\mathcal{M}, s_i \models \varphi$.
9. $\mathcal{M}, s \models AG\ \varphi$ if and only if for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$, and for all $s_i$ along the path, we have $\mathcal{M}, s_i \models \varphi$.
10. $\mathcal{M}, s \models EF\ \varphi$ if and only if there is a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$ and for some $s_i$ along the path we have $\mathcal{M}, s_i \models \varphi$.
11. $\mathcal{M}, s \models AF\ \varphi$ if and only if for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$ there is some $s_i$ such that $\mathcal{M}, s_i \models \varphi$.
12. $\mathcal{M}, s \models E[\varphi_1 U \varphi_2]$ if and only if there exists a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$ and for some $s_i$ along the path $\mathcal{M}, s_i \models \varphi_2$ and $\forall j, j < i\ \ \mathcal{M}, s_j \models \varphi_1$.
13. $\mathcal{M}, s \models A[\varphi_1 U \varphi_2]$ if and only if for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$ there exists $s_i$ along the path such that $\mathcal{M}, s_i \models \varphi_2$ and $\forall j, j < i\ \ \mathcal{M}, s_j \models \varphi_1$.
14. $\mathcal{M}, s \models E[\varphi_1 B \varphi_2]$ if and only if there exists a path $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$ and for some $s_i$ along the path $\mathcal{M}, s_i \models \varphi_2$ implies $\exists j, j < i\ \ \mathcal{M}, s_j \models \varphi_1$.

15. $\mathcal{M}, s \models A[\varphi_1 B \varphi_2]$ if and only if for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1 = s$ there exists $s_i$ along the path such that $\mathcal{M}, s_i \models \varphi_2$ implies $\exists j, j < i \;\; \mathcal{M}, s_j \models \varphi_1$.

We say an *elementary formula* is a formula of the form $p$, $\neg p$, $EX\varphi$ or $AX\varphi$ where $p$ is an atomic proposition and $\varphi$ is a CTL formula. The classification of non-elementary formulae is shown in Table 1 using Smullyan's $\alpha$- and $\beta$-notation.

Table 1: Smullyan's $\alpha$- & $\beta$-notation for CTL

| $\alpha$ | $\alpha_1$ | $\alpha_2$ |
|---|---|---|
| $\varphi \wedge \psi$ | $\varphi$ | $\psi$ |
| $EG\ \varphi$ | $\varphi$ | $EX\ EG\ \varphi$ |
| $AG\ \varphi$ | $\varphi$ | $AX\ AG\ \varphi$ |
| $E[\varphi B \psi]$ | $\neg\psi$ | $\varphi \vee EX\ E[\varphi B \psi]$ |
| $A[\varphi B \psi]$ | $\neg\psi$ | $\varphi \vee AX\ A[\varphi B \psi]$ |

| $\beta$ | $\beta_1$ | $\beta_2$ |
|---|---|---|
| $\varphi \vee \psi$ | $\varphi$ | $\psi$ |
| $EF\ \varphi$ | $\varphi$ | $EX\ EF\varphi$ |
| $AF\ \varphi$ | $\varphi$ | $AX\ AF\ \varphi$ |
| $E[\varphi U \psi]$ | $\psi$ | $\varphi \wedge EX\ E[\varphi U \psi]$ |
| $A[\varphi U \psi]$ | $\psi$ | $\varphi \wedge AX\ A[\varphi U \psi]$ |

## 3.3 Tableau-based satisfiability checking for CTL

Tableau systems obey the subformula principle - all formulae occurring in a tableau proof are subformulae of the formula being proved. Subformulae are obtained by applying a set of rules based on the semantics of the particular logic. Applications of these rules forms a tree, called the tableau. For propositional logic, tableau-based procedures include the following steps: to show a formula $\varphi$ is valid we try to show its negation $\neg\varphi$ is not satisfiable, i.e., there is no assignment of truth values to propositional variables in $\neg\varphi$ to make it *true*. The expression $\neg\varphi$ is decomposed into subformulae by applying tableau expansion rules. If a branch of the tableau contains a pair of contradictory formulae (i.e., $\psi$ and $\neg\psi$), then this branch is marked as *closed*. The tree construction stops when all the branches close or there is no other formula to which a tableau rule can be applied. An open branch in a completed tableau (rules have been applied to all the formulae) gives a counter example, i.e, an assignment which satisfies $\neg\varphi$. If all branches close, the tableau is said to be *closed* and $\varphi$ is declared valid.

In CTL tableau, Boolean connectives are handled the same way as in propositional logic, and temporal connectives are handled by decomposing them into a requirement on the "current state" and a requirement on "the rest of the sequence" [25]. Implementing the tableau rules will cause some branches of the tableau to loop forever. However, the tableau can be made finite by identifying nodes containing the same set of formulae. A formula $\varphi$ of the form $EF\varphi, AF\varphi, E[\varphi\ U\ \psi]$, or $A[\varphi\ U\ \psi]$ is called an *eventuality formula* [1]. Eventuality formulae state "something will happen eventually in the future". To guarantee validity of a CTL formula $\varphi$, in addition to checking for propositional inconsistencies, we need to check for unsatisfiability of all the eventuality formulae. The tableau rules for CTL can be categorized into four categories: the $\alpha$ rules, the $\beta$ rules, the $X$ rule, and the terminal rules. The $\alpha$ rules are for the conjunctive operators (i.e., $\wedge$, $EG$, $AG$, $EB$, and $AB$) and each creates one child (e.g., the rule for $EG\varphi$ creates a child with the formulas $\varphi$ and $EXEG\varphi$). The $\beta$ rules are for the disjunctive operators (i.e., $\vee$, $EF$, $AF$, $EU$ and $AU$) and each creates two children (e.g., the rule for $AF\varphi$ creats a child with $\varphi$ and a child with $AXAF\varphi$). The $X$ rule is for the $X$ operator and the number of children created is dependent on the number of $EX$ formulae in a node. The $X$ rule states that if there is $\{EX\phi_1, \cdots, EX\phi_n, AX\psi_1, \cdots, AX\psi_m\}$ in a node then there are $n$ children of the node, with $\{\phi_1, \psi_1, \cdots, \psi_m\}$ at child 1, $\{\phi_2, \psi_1, \cdots, \psi_m\}$ at child 2, and so on.

There are two terminal rules, one (known as the *id* rule) is applied when there is a contradiction on a branch and the other (known as the *block* rule) is applied when expanding a branch causes a loop. If a node $m$ is about to be created as a child of a node $n$ and there is an ancestor $n_0$ of $n$ having the same set of formulae in $m$, then $m$ is not created, and $n$ and $n_0$ are connected with a feedback edge (these situations causes loops in the tableau).

A Hintikka structure for $\varphi$ is a finite partial representation of a model for $\varphi$. A formal discussion on Hintikka structures for CTL formulae can be found in [1], [2]. The tableau procedure is a systematic search for a Hintikka structure; to determine the satisfiability of $\varphi$ the tableau shows there is no Hintikka structure for $\neg\varphi$.

The two-pass tableau-based decision procedures [2], [3] test the satisfiability of a CTL formula $\varphi$ in two steps or "passes". In the first step, it constructs a tableau $\mathcal{T}_\varphi$, for $\varphi$, by applying tableau construction rules. If any Hintikka structure satisfies $\varphi$, then there is at least one represented by $\mathcal{T}_\varphi$ [8]. In the second step, inconsistent nodes (nodes that cannot be a part of any Hintikka structure for $\varphi$) are identified. The second pass uses an algorithm known as the marking algorithm to mark the inconsistent nodes; the marking algorithm marks the root of the tableau if there is no Hintikka structure for the input formula [2]. Termination of the one-pass tableau procedure is guaranteed [11].

The one-pass tableau algorithm for CTL [1] uses a single pass to determine the satisfiability of a formula. Instead of constructing the tableau first and then finding the Hintikka structure, the one-pass tableau determines the existence of a Hintikka structure while constructing the tableau, through the use of a *history* and a *variable* associated with each tableau node. In the one-pass tableau, loops are determined by looking at the *history* of the current node; the *history* is passed from parent to child. The *variable* propagates information about the unsatisfiable eventualities from a child to its parent. The history of a node is calculated while applying a tableau rule. The variable of a terminal node is calculated according to the terminal rule when a branch of the tableau terminates, and propagated upward.

The tableau rules for the one-pass tableau for satisfiability checking are available in [1]. A tableau node, in a one-pass tableau, contains three components - a set of formulae $\Gamma$, a history *Fev* and *Br*, and a variable *uev*, the three are $\Gamma :: Fev, Br :: uev$, where the symbol "::" separates the three components. Here, *Fev* keeps track of the satisfiable eventualities, *Br* keeps track of the formulae that may create loops and is used by one of the terminal rules to identify loops, and *uev* keeps track of the unsatisfiable eventualities. The *uev* of a node is set to $\{(false,m)\}$ if both branches created by a $\beta$ rule are *closed* due to propositional inconsistencies; the *uev* is set to the empty set if there are no unsatisfiable eventualities or propositional inconsistencies in any of the children; the *uev* is set to the set of all the unsatisfiable eventualities of the children if both children have unsatisfiable eventualities; and finally the *uev* is set to the set of all unsatisfiable eventualities of a child if either of the two children has unsatisfiable eventualities.

The one-pass procedure starts with the negation of the formula of interest (the input formula) in negation normal form (NNF) as the root and the rules are applied on the root to construct the tableau. If, after the construction, the *uev* of the root is not empty, we call it a closed tableau. A closed tableau means the negation of the input formula is not satisfiable, i.e., the input formula is valid. On the other hand, if the uev of the root is the empty set we say the tableau is open, meaning the negation of the input formula is satisfiable, and the input formula is not valid. We discuss one of the rules, namely the $AF$ rule below. A detailed discussion of the one-pass tableau rules may be found in [11].

$$(AF) \quad \frac{AF\varphi; \Gamma :: Fev, Br :: uev}{\varphi; \Gamma :: \{AF\varphi\} \cup Fev, Br :: uev_1 \mid AXAF\varphi; \Gamma :: Fev, Br :: uev_2}$$

The *uev* of the parent is calculated from the $uev_1$ and $uev_2$ of the children as follows:

$$uev = \begin{cases} uev_1 & \text{if } uev_2 = \{(false, m)\} \\ uev_2 & \text{if } uev_1 = \{(false, m)\} \\ \{(AF\varphi, n)\} & \text{otherwise} \end{cases}$$

Here, $n = max\big(f(AF\varphi, uev_1) \cup f(AF\varphi, uev_2)\big)$. The function $f(AF\varphi, uev')$ returns the index of $AF\varphi$ in *uev'*. We show an example of the $AF$ rule in Fig. 1. In node $n_6$, $Fev$ and $Br$ came from its predecessor (not shown in the figure). An application of the $AF$ rule on $n_6$ creates nodes $n_7$ and $n_8$. The *id* rule is applied on $n_7$ and the *block* rule is applied on $n_8$. We show $n_6$ with the value of *uev* calculated from its children. For $n_6$, $uev_1 = \{(false, 1)\}$ and $uev_2 = \{(AF\varphi, 0)\}$. As $uev_1 = \{(false, 1)\}$, the *uev* of $n_6$ is set to $uev_2$ which is $\{(AF\varphi, 0)\}$.

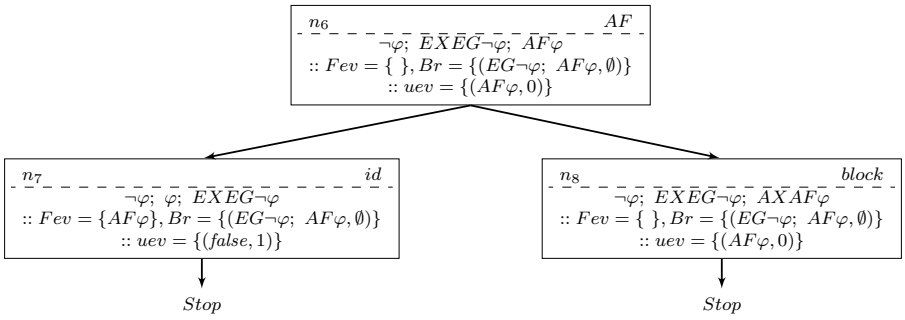


Figure 1: Illustration of the AF rule.

# 4 The One-Pass Tableau-Based Model Checking

A generic framework using the tableau method for model checking is described in [9]. The main idea of tableau-based model checking is that given a Kripke structure $\mathcal{M}$ with initial state $s_0$ and a property $\varphi$, the algorithm simulates the construction of the tableau in $\mathcal{M}$. The construction starts from $s_0$ and moves forward along the transitions in $\mathcal{M}$. More specifically,

1. The tableau construction starts with $(\neg\varphi, s_0)$ where $s_0 \in S$.
2. The tableau construction is similar to the tableau for satisfiability. However, the construction rules must be modified so that the tableau nodes are properly associated with the states of $\mathcal{M}$.
3. When the tableau construction is completed, $\mathcal{M}$ satisfies $\varphi$ iff the final tableau with $(\neg\varphi, s_0)$ is a closed tableau.

We show the modifications required to use the one-pass tableau procedure to perform model checking.

**Definition 3** (Closure of a formula $cl(\varphi)$ in CTL)**.** *The closure $cl(\varphi)$ of a formula $\varphi$ is the least set of formulae such that:*

*1.* $\top, \varphi \in cl(\varphi)$*;*

2. *$cl(\varphi)$ is closed under taking subformulae;*

3. *if $\psi \in cl(\varphi)$ and $\psi$ does not begin with $\neg$ then $\neg\psi \in cl(\varphi)$;*

4. *$cl(\varphi)$ is closed under taking all components of $\alpha-$formulae and $\beta-$formulae.*

Given a Kripke structure $\mathcal{M}$, and a CTL formula $\varphi$, $L[cl(\varphi), s] := \big(L(s) \cup \{\neg p \mid p \in AP \setminus L(s)\}\big) \cap cl(\varphi)$. It can be shown that $L[cl(\varphi), s]$ consists of a set of atomic propositions and the negation of atomic propositions only. For example, given the Kripke structure in Figure 2, and a CTL formula $\varphi = AG(p \rightarrow AFq)$, $L[cl(\varphi), s] = \{p, q\}$.

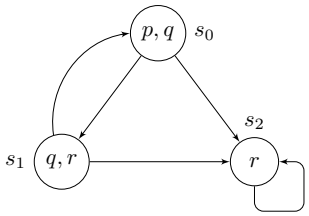


Figure 2: A simple Kripke structure with three states.

In the one-pass tableau-based model checking, the tableau construction starts with $(s_0, \neg\varphi \cup L[cl(\neg\varphi), s_0])$. Here, $s_0$ is the initial state of the model and $\varphi$ is the CTL property to be verified. In model checking, all the tableau rules are same as the tableau for satisfiability checking except for the $X$ rule. The $X$ rule which deals with formulae referring to the "next state", is modified to denote a transition from one state to another. The next states of a state can be identified from the given transition relation of the Kripke structure. The one-pass tableau model checking algorithm given in Algorithm 1. The tableau construction starts from the initial states and to reduce the branching, the $\alpha$ rules are applied before the $\beta$ rules. The $\alpha$ and $\beta$ rules do not make transitions from one state to another. The $X$ rule is applied when a tableau node has only $EX$ and $AX$ formulae to which to apply tableau rules. In the tableau-based satisfiability checking, the number of children of a node having $EX$ and $AX$ formulae depends on the number of $EX$ formulae in the node. In model checking, the number of children is determined from the number of $EX$ formulae and the number of states adjacent to the current state. If the constructed tableau is closed then the property $\varphi$ is declared *true*; otherwise, an open branch of the tableau shows a counter example.

## 4.1 System description

We implemented the tableau-based model checking framework using the C++ programming language. We used the CPN tool to design the Petri net workflow models. The CPN tool stores the Petri net in the Petri net markup language (PNML) which is an XML-based interchange format for Petri nets. Our framework could be modified to deal with any Petri net graphical editor with an XML based interchange format. The top-level structure of the tableau-based model checking framework is shown in Fig. 3 and a brief discussion of each component is given below.

**The XML Parser:** Most of the information in the XML file generated by the CPN tool is editor specific information, such as: the position of the nodes, the size of the nodes, and colours of different parts, etc. Our XML Parser reads the input PNML file from the beginning to the

**Algorithm 1** Given a *Kripke structure* and a *property*, this algorithm generates a tableau using the one-pass tableau procedure

*root* $\leftarrow$ *tableauNode(property* $\cup$ $L[cl(property), state]$ *, state)* {Here, *tableauNode*($\varphi$, $s_0$) is a constructor that creates a `tableauNode` with $\varphi$ in the `formulaList` and $s_0$ in the `stateSpaceID`}
*nodeStack.push(root)*
**while** one-pass tableau rules have not been applied to all the nodes in *nodeStack* **do**
  *tempTableauNode* $\leftarrow$ *nodeStack.pop()*
  **if** the id rule is applicable to *tempTableauNode* **then**
    apply the id rule
  **else if** a linear rule is applicable to *tempTableauNode* **then**
    *newTableauNode* $\leftarrow$ *tableauNode*($\{\alpha_1, \alpha_2\} \cup \Gamma$, $s_i$) {Here, $\Gamma$ is the set of formulae in *tempTableauNode* and $s_i$ is the `the stateSpaceID` in *tempTableauNode*}
    *nodeStack.push(newTableauNode)*
  **else if** a universal branching rule is applicable to *tempTableauNode* **then**
    *newTableauNode1* $\leftarrow$ *tableauNode*($\{\beta_1\} \cup \Gamma$, $s_i$)
    *nodeStack.push(newTableauNode1)*
    *newTableauNode2* $\leftarrow$ *tableauNode*($\{\beta_2\} \cup \Gamma$, $s_i$)
    *nodeStack.push(newTableauNode2)*
  **else if** an existential branching rule is applicable to *tempTableauNode* **then**
    *adjList* $\leftarrow$ all the states adjacent to $s_i$ in the *Kripke structure*
    **for all** $s_j \in$ *adjList* **do**
      *newTableauNode* $\leftarrow$ *tableauNode*($\Delta \cup \psi \cup L[cl(property), s_j]$, $s_j$) {Here, *tempTableauNode* has a formula of the form $EX\psi; AX\Delta$}
      *nodeStack.push(newTableauNode)*
    **end for**
  **else**
    apply the block rule
  **end if**
**end while**

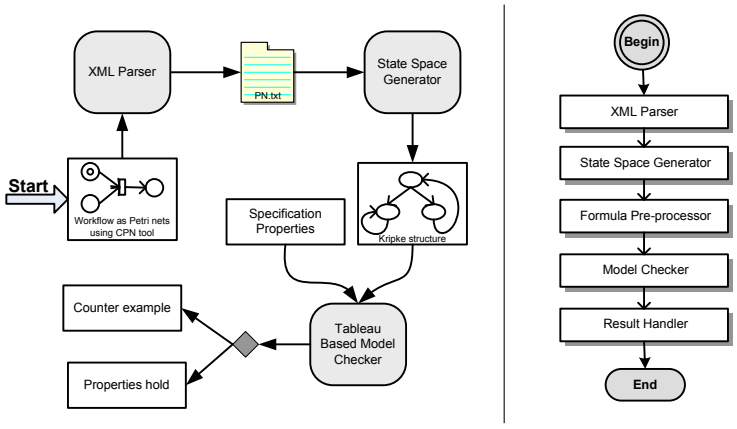


Figure 3: The components of the tableau-based model checking framework.

end and extracts the information related to the Petri net places and transitions and stores, and stores it in a simple text file.

**The State Space Generator:** The State Space Generator loads the Petri net model from the simple text file generated by the XML Parser. The Petri net is represented in the memory as a list of transitions. Two types of information control a transition firing, a guard and an action. If we have a loop in the workflow model then a set of transitions are fired until the guard becomes *false*. A guard represents a condition of the from *variable op value*, where $op \in \{==, != , <, >, \leq, \geq\}$. An action represents an assignment of the form *variable = exp*, where *exp* is a *variable* or of the form *variable* $op_1$ *value*, and $op_1 \in \{+, -, *\}$. A transition can have an action and/or a guard, whereas a place can have only an action. Actions associated with transitions can change the value of a variable in the output places. After loading the Petri net model from the text file, the Kripke structure is generated by simulating the Petri net[11].

**The Formula Pre-processor:** The Formula Pre-processor module applies four pre-processing steps before applying the tableau model checking algorithm. The first step is to rewrite the $U$ and $B$ operators in the given property. For example, a formula of the form $E[\varphi\ U\ \psi]$ is written as $(\varphi\ EU\ \psi)$, this makes it easier to generate a parse tree for the formula. The second pre-processing step is to generate a parse tree for the property. Using parse trees provides two benefits: we can easily identify the first operator to apply a tableau rule to (it is the root of the parse tree) and easily identify the subformulae. The third pre-processing step is to change the input property $\varphi$ to $\neg\varphi$. The final pre-processing step is to convert $\neg\varphi$ to NNF.

**The Model Checker:** The Model Checker module uses two functions − a state space handler to manage the Kripke structure and a property handler to manage the property parse tree. The Model Checker module implements the one-pass tableau model checking algorithm according to the previous discussion.

**The Result Handler:** The purpose of the Result Handler module is to show the output of the model checker in a readable format. Currently, our model checker can show the output as a list of tableau nodes which can be transformed into a tree structure by hand. If a property does not hold, we can get a counter example by investigating an open branch.

First, we implemented the one-pass tableau algorithm for satisfiability checking and tested our implementation with a comprehensive set (41 in total) of CTL formulae available at [17]; see the results in [11]. Then we modified the one-pass tableau algorithm for model checking. We used the Mahone cluster (a parallel cluster of 134 nodes with 64GB RAM per node) of ACEnet [2] to run our experiments.

# 5 Case Study

Our research is part of a collaboration among academic researchers, an industry partner and the local health authority to develop innovative workflow tools for health services delivery [13], [6]. Healthcare workflows are developed from guidelines or best practises defined by healthcare professionals. Such guidelines are processes describing the activities for providing treatment to a patient. Using the CPN tool, we modeled a workflow following the national Hospice Palliative Care (HPC) guideline [3] and used our tool model check some properties, some of which are listed

[2] ACEnet: http://www.ace-net.ca
[3] Canadian HPC association: http://www.chpca.net/

below.

HPC refers to the comfort care that reduces the severity of a disease rather than providing a cure. For example, if surgery cannot be performed to remove a tumour, radiation treatment might be tried to reduce its rate of growth, and pain management could help the patient manage physical symptoms. The HPC guideline containing 51 tasks is depicted in Fig. 4; the Petri net model contains 55 places and 51 transitions, and the corresponding workflow state space graph consists of 67 states. We verified the following properties of the model:

**Property 1:** $AF end\_of\_workflow$
The `end_of_workflow` will always be reached.

**Property 2:** $AG(error\_in\_therapy \rightarrow EF report\_to\_supervisor)$
Any error in therapy is always reported to the supervisor.

**Property 3:** $AG(prepare\_care\_plan \rightarrow AF present\_care\_plan)$
After a care plan is prepared, it is always presented to the patient.

**Property 4:** $AG\neg(\neg define\_limits\_of\_conf. \wedge share\_accurate\_info)$
Limits of confidentiality are always defined before information is shared.

The verification results are summarized in Table 2. Further experiments may be found in [11], including experiments on verification of large models, and an example of a smaller model with a failed property and the output of a counter model. In our current implementation the output of counter models for large Petri nets is difficult for the human eyes to read.

Table 2: Property verification results of the one-pass tableau model checker

| **Property** | **Time** (in sec) | **No. of tableau nodes** | **Memory** (in MB) | **Valid** |
|---|---|---|---|---|
| Property 1 | 3.887 | 2266 | 8.1 | Yes |
| Property 2 | 13.931 | 4968 | 16.5 | Yes |
| Property 3 | 12.389 | 4778 | 15.9 | Yes |
| Property 4 | 11.399 | 4920 | 16.4 | Yes |

# 6 Conclusion and Future work

In this paper we presented a model checking framework for workflow model verification. We used the one-pass tableau method for model checking which can efficiently verify properties for a large workflow model. The framework can be improved and extended easily as the architecture can be adapted to support different workflow modelling languages (e.g., YAWL [24]) as well as a variety of property specification languages, such as LTL, timed CTL, other modal logics such as BDI logics (logics for beliefs, desires, and intentions) as needed to verify various aspects of large scale enterprise information systems. The latter modifications in general simply require the addition and/or replacement of one-pass tableau rules. While there are other approaches to model checking Petri nets (e.g., LoLa [18] and Fast [5]) these in general lack flexibility as they use a fixed specification language for property specifications. We presented an implementation of the framework, but a number of improvements are possible. The tree structure, generated

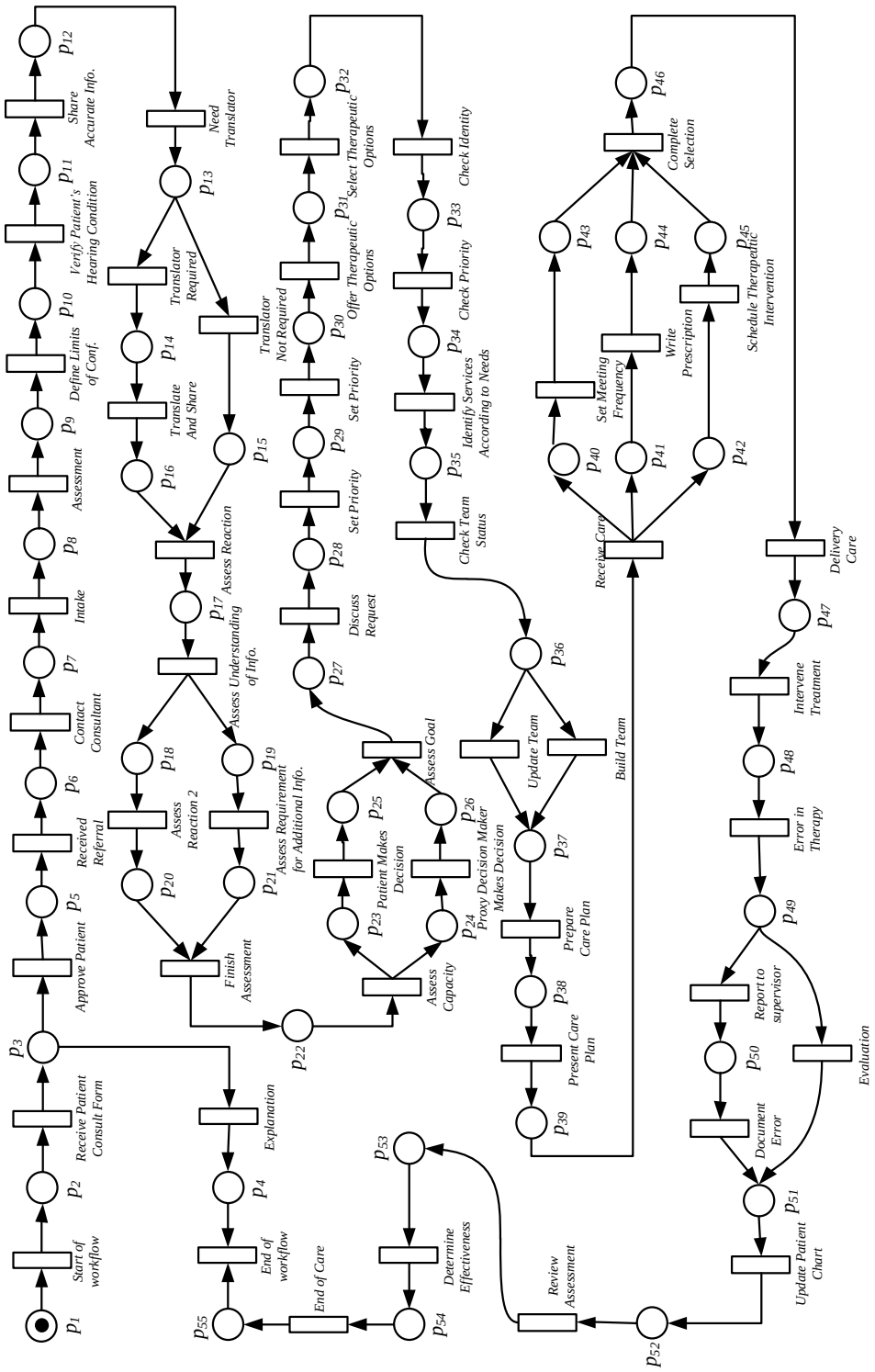


Figure 4: The HPC model using Petri nets.

by the tableau model checking algorithm, is not possible to show on the console output. A graphical user interface showing the tableau would help the user better analyse the counter models. Another improvement would be to apply high performance computing techniques to increase the efficiency of the model checking algorithm. In the one-pass tableau method, only one branch of the derivation tree needs to be considered at any stage, making it suitable to implement on a bank of parallel processors [1]. In [21], [12], the authors discussed two approaches to parallel temporal tableau for LTL using the two-pass tableau procedure; these need investigation for the one-pass method. The first approach applies parallelism by dividing the sequential algorithm into separate sub-problems and distributing the sub-problems to different processors. In this approach communication between the processes are maintained using two shared queues. The second approach does not use any shared queue, hence the inter-process communication increases. However, dividing into sub-problems does not ensure equal load distribution across the processors, because one sub-problem may take less time than the others, i.e., some processors will remain idle while the others are working. The inter-process communication increases in the second approach. In literature, experiments show that the second approach performs better than the first approach for LTL [21]. Many optimization techniques including unit propagation, simplification, and backjumping have been developed for tableau-based modal logic systems which can be applied to the one-pass tableau-based model checking algorithm to enhance performance. Extensions of the method to include timing information are fairly straightforward [15].

**Acknowledgements:** The authors were supported by an NSERC Discovery Grant, by the Atlantic Canada Opportunities Agency, and by an ACEnet Graduate Research Fellowship and benefited from many discussions with clinicians from the Guysborough Antigonish Strait Health Authority (GASHA).

# References

[1] P. Abate, R. Goré, and F. Widmann. One-Pass Tableaux for Computation Tree Logic. In *Proc. of the 14th Int. Conference on Logic for Programming, Artificial Intelligence and Reasoning*, LPAR'07, pages 32–46, 2007.

[2] M. Ben-Ari, A. Pnueli, and Z. Manna. The Temporal Logic of Branching Time. In *Proc. of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages(POPL '81)*, volume 20, pages 164–176, 1981.

[3] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Proc. of Logic of Programs, Workshop*, volume 131, pages 52–71, 1982.

[4] J. Dallien, W. MacCaull, and A. Tien. Initial Work in the Design and Development of Verifiable Workflow Management Systems and Some Applications to Health Care. In *Proc. of the 5th Int. Workshop on Model-based Methodologies for Pervasive and Embedded Software, Co-located with ETAPS 2008*, pages 78–91, 2008.

[5] Laboratoire Spécification et Vérification (LSV). `http://www.lsv.ens-cachan.fr/Software/fast/`.

[6] Centre for Logic and Informatoin (CLI). `http://logic.stfx.ca/`.

[7] S. Giro. Workflow Verification: a New Tower of Babel. In *Proc. of the Int. Modelling and Simulation Multiconference (IMSM 2007)*, 2007.

[8] V. Goranko, A. Kyrilov, and D. Shkatov. Tableau Tool for Testing Satisfiability in LTL: Implementation and Experimental Analysis. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 262, 2010.

[9] Valentin Goranko. Temporal Logics for Specification and Verification. In *Proc. of the European Summer School in Logic, Language and Information (ESSLI'09)*, 2009.

[10] V. Gruhn and R. Laue. Using Timed Model Checking for Verifying Workflows. In *Proc. of Computer Supported Activity Coordination (2005)*, pages 75–88, 2005.

[11] M. Z. Islam. A tableau-based workflow verification framework for computation Tree Logic (CTL). Master's thesis, St. Francis Xavier University, Antigonish, Canada, 2012.

[12] R. Jonathon. A Blackboard Approach to Parallel Temporal Tableaux. In *In Proc. of Artificial Intelligence, Methodologies, Systems and Applications*, 1994.

[13] W. MacCaull, H. Jewers, and M. Latzel. Using an Interdisciplinary Approach to Develop a Knowledge-driven Careflow Management System for Collaborative Patient-centred Palliative Care. In *In Proc. of the 1st ACM Int. Conference on Health Informatics*, 2010.

[14] Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary, 1999. Document Number WFMC-TC-1011.

[15] K. Miller and W. MacCaull. Verification of Careflow Management Systems with Timed $\text{BDI}_{CTL}$ Logic. In *Proc. of Int. Workshops on Business Process Management (BPM 2009)*, volume 43, pages 623–634, 2010.

[16] R. Müller, U. Greiner, and E. Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering*, 51:223–256, 2004.

[17] Research School of Information Sciences and Engineering (RSISE). The Tableau Workbench. `http://twb.rsise.anu.edu.au/demolist/`.

[18] Theory of Programming Languages and Programming. `http://www.informatik.uni-rostock.de/tpp/lola/`.

[19] F. Rabbi, H. Wang, and W. MacCaull. YAWL2DVE: An Automated Translator for Workflow Verification. In *Proc. of the 4th IEEE Int. Conference on Secure Software Integration and Reliability Improvement (SSIRI '10)*, pages 53–59, 2010.

[20] W. Sadiq and M. E. Orlowska. Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. In *Proc. of the 11th Int. Conference on Advanced Information Systems Engineering (CAiSE '99)*, pages 195–209, 1999.

[21] R. I. Scott, M. D. Fisher, and J. A. Keane. Parallel Temporal Tableaux. In *In Proc. of the 4th Int. Euro-Par Conference on Parallel Processing*, 1998.

[22] M. C. Stolz. Verification of Workflow Control-Flow Patterns with the SPIN Model Checker. Master's thesis, University of Bern, Switzerland, 2010.

[23] The CPN tool website. `http://www.cpntools.org/`.

[24] W. M. P. van der Aalst and A. ter Hofstede. YAWL: yet another workflow language. *Journal of Information Systems*, 30(4):245–275, 2005.

[25] P. Wolper. The Tableau Method for Temporal Logic: An Overview. *Logique et Analyse, 28(110–111)*, pages 119–136, 1985.

[26] L. Zeng, D. Flaxer, H. Chang, and J. J. Jeng. $\text{PLM}_{flow}$-Dynamic Business Process Composition and Execution by Rule Inference. In *Proc. of the 3rd Int. Workshop on Technologies for E-Services (TES '02)*, pages 141–150, 2002.