Fair Cycle Detection using Description Logic Reasoning

Shoham Ben-David[1], Jeffrey Pound[1], Richard Trefler[1], Dmitry Tsarkov[2], Grant Weddell[1]

1. David R. Cheriton School of Computer Science, University of Waterloo
2. School of Computer Science, University of Manchester

1 Introduction

Model checking ([12, 20], c.f.[11]) is a technique for verifying finite-state systems, that has been proven to be very effective in the verification of hardware and software programs. In model checking, a model M, given as a set of state variables V and their next-state relations, is verified against a temporal logic formula φ. When φ holds on all computation paths of M we write $M \models \varphi$. The most commonly used temporal logics are Linear Temporal Logic (LTL) [19] and Computation Tree Logic (CTL) [12].

Temporal logic specifications, whether given in LTL or in CTL, are roughly divided into two basic categories [16, 1]: formulas that specify *safety* properties and formulas that specify *liveness* properties. Informally, a safety formula states that "something *bad* never happens", and a violation of it can be shown by a finite prefix of a computation path, reaching a bad state. A liveness formula asserts that "something *good* will eventually happen", and a counterexample for it must contain an infinite path where a good state never appears (a *cycle*, in case of a finite model). A simple example of a liveness property is the CTL formula AFp asserting that the proposition p must appear eventually on every computation path of the model. In many cases, a liveness formula φ is accompanied by a set of *fairness constraints*: Boolean events that must appear infinitely often on a path to make it *fair*. When fairness constraints are present, φ should only be verified on fair paths, and a counterexample should demonstrate a *fair cycle*: an infinite computation path on which φ fails to hold, but each fairness constraint appears infinitely often.

Symbolic model checking is performed using two main approaches. The first is based on BDDs (e.g. SMV [17]), and the second is based on satisfiability solving (SAT) technology [6]. For both methods, liveness formulas are considered more difficult to verify than safety ones. Special attention has been devoted to detecting fair cycles in recent years, both using BDDs methods [7, 8], and using SAT-based techniques [2, 14].

We consider a different approach for fair cycle detection, that makes use of Description Logic (DL) technology. We cast a model M and the negation of the given liveness specification φ as a satisfiability query in DL, such that if an interpretation is found, it indicates an error in M. While a model and a liveness formula can be easily encoded as a terminology in $\mathcal{ALC}$, this is not the case with fairness constraints. In order to express fairness, more expressive dialects are needed (see, for example [15, 10]). We propose a modification to the $\mathcal{ALC}$ reasoning technique based on tableau construction, that allows us to detect a fair cycle. When constructing a tableaux, cycles are represented by *blocking*. A node x is said to be blocked by a node y if there exists a path of nodes from y to x, and the label of x is a subset of the label of y. Given a cycle in the tableau, and a fairness constraint FC, we attempt to make the cycle fair by adding FC to the label of a node in the cycle. To accomplish this, we extend the tableaux algorithm with a new rule that we call a *fairness rule*. We show that our method is sound and terminating, and discuss how completeness can be achieved.

We have implemented our method in the Description Logic reasoner FaCT++ [22], and we present experimental results comparing our method with runs using the model checker VIS [9]. Although running on a few examples only, the results demonstrate the potential of our method, as they significantly outperform VIS on some of the examples.

The rest of the paper is organized as follows. In the next section we give the necessary definitions, and in Section 3 we present the translation of a liveness model checking problem into a DL satisfiability query. Section 4 is the main section of the paper, where we present our algorithm for fair cycle detection. Section 5 presents experimental results, and Section 6 concludes the paper.

2 Background and Definitions

2.1 Description Logic

Definition 1 (Description Logic $\mathcal{ALC}$) *Let* NC *and* NR *be disjoint sets of* atomic concepts $\{A_1, A_2, \ldots\}$ *and* atomic roles $\{R_1, R_2, \ldots\}$ *respectively. The set of* concepts C *is the smallest set including* NC *such that if* $C, D \in \mathsf{C}$ *and* $R \in \mathsf{NR}$*, then so are* $\neg C, C \sqcap D$ *and* $\exists R.C$.

Additional concepts are defined as syntactic sugaring of those above:
• $C \sqcup D = \neg(\neg C \sqcap \neg D)$ • $\forall R.C = \neg \exists R.\neg C$ and • $\top = A \sqcup \neg A$ for some atomic concept A.

A *general concept inclusion* (GCI) is an expression of the form $C \sqsubseteq D$, where C and D are arbitrary concepts. A *terminology* (or TBox) $\mathcal{T}$ consists of a finite set of concept inclusions.

The *semantics* of expressions is defined with respect to a structure $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, called an *interpretation*, where $\Delta^{\mathcal{I}}$ is a non-empty set of individuals, and $(\cdot)^{\mathcal{I}}$ is an interpretation function that maps atomic concepts A to a subset of $\Delta^{\mathcal{I}}$ and atomic roles R to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended to arbitrary concepts in a way that satisfies each of the following:

– $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$,
– $(\exists R.C)^{\mathcal{I}} = \{e \in \Delta^{\mathcal{I}} : \exists (e, e') \in R^{\mathcal{I}}$ s.t. $e' \in C^{\mathcal{I}}\}$, and
– $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$.

An interpretation $\mathcal{I}$ satisfies a GCI $(C \sqsubseteq D)$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and a TBox $\mathcal{T}$ if it satisfies each concept inclusion in $\mathcal{T}$.

The *concept satisfiability problem* is to determine, for a given TBox $\mathcal{T}$ and concept C, if there exists an interpretation $\mathcal{I}$ that satisfies $\mathcal{T}$ and for which $C^{\mathcal{I}}$ is non-empty, written $\mathcal{T} \models_{dl} C$.

Tableaux Algorithms for Concept Satisfiability in $\mathcal{ALC}$. The tableaux algorithm works on a labeled tree, called a *completion tree*, that has a close correspondence to an interpretation. For a concept C, we write $\mathsf{nnf}(C)$ to denote the Negation Normal

Form (NNF) of C, write $\dot{\neg}C$ to denote the NNF of $\neg C$, and write $\mathsf{sub}(C)$ to denote the set of all subconcepts of C (including C) and their negation. For a TBox $\mathcal{T}$ we define $\mathsf{sub}(\mathcal{T}) = \bigcup_{(C \sqsubseteq D) \in \mathcal{T}} \mathsf{sub}(C) \cup \mathsf{sub}(D)$.

Definition 2 *Let $\mathcal{T}$ be an $\mathcal{ALC}$ TBox and C is a concept in NNF. A* completion tree *for C with respect to $\mathcal{T}$ is a directed graph $\mathbf{G} = (V, E, \mathcal{L})$ where each node $x \in V$ is labelled with a set $\mathcal{L}(x) \subseteq \mathsf{sub}(\mathcal{T}) \cup \mathsf{sub}(C)$ and each edge $\langle x, y\rangle \in E$ is labelled with a role name $\mathcal{L}(\langle x, y\rangle) \in R_N$.*

If $\langle x, y\rangle \in E$, then y is called a successor *of x and x is called a* predecessor *of y. If, in addition, $R = \mathcal{L}(\langle x, y\rangle)$, then y (x) is called an R-*successor *(R-*predecessor*) of x (y).* Ancestor *is the transitive closure of predecessor, and* descendant *is the transitive closure of successor.*

$\mathbf{G}$ *is said to contain a* clash *if for some $A \in \mathsf{NC}$ and node x of $\mathbf{G}$, $\{A, \neg A\} \subseteq \mathcal{L}(x)$.*

The tableaux algorithm for checking concept satisfiability of C w.r.t. $\mathcal{T}$ starts with the completion tree $\mathbf{G} = (\{r_0\}, \emptyset, \mathcal{L})$ where $\mathcal{L}(r_0) = \{\mathsf{nnf}(C)\}$. $\mathbf{G}$ is then expanded by repeatedly applying the expansion rules given in Figure 1, stopping if a clash occurs. In order to ensure termination we need to restrict the creation of new nodes in the

$\sqsubseteq$-rule: if 1. $C_1 \sqsubseteq C_2 \in \mathcal{T}$, and
  2. $\{\dot{\neg}C_1, \mathsf{nnf}(C_2)\} \cap \mathcal{L}(x) = \emptyset$
 then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{\dot{\neg}C_1, \mathsf{nnf}(C_2)\}$

$\sqcap$-rule: if 1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, and
  2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
 then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C_1, C_2\}$

$\sqcup$-rule: if 1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$, and
  2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
 then set $\mathcal{L}(x) = \mathcal{L}(x) \cup \{C\}$ for some $C \in \{C_1, C_2\}$

$\exists$-rule: if 1. $\exists R.C \in \mathcal{L}(x)$, x is not blocked, and
  2. x has no R-successor y with $C \in \mathcal{L}(y)$,
 then create a new node y with $\mathcal{L}(\langle x, y\rangle) = R$
  and $\mathcal{L}(y) = \{C\}$

$\forall$-rule: if 1. $\forall R.C \in \mathcal{L}(x)$, and
  2. there is an R-successor y of x such that $C \notin \mathcal{L}(y)$
 then set $\mathcal{L}(y) = \mathcal{L}(y) \cup \{C\}$

Fig. 1. Tableaux expansion rules for $\mathcal{ALC}$

completion tree. The notion of *blocking* is used for this purpose.

Definition 3 (Blocking) *A node x is* label blocked *if it has an ancestor y such that $\mathcal{L}(x) \subseteq \mathcal{L}(y)$. In this case, we say that y* blocks *x. A node is* blocked *if either it is label blocked or its predecessor is blocked.*

When nodes in a branch of the completion tree resemble ancestor nodes, a block is established to ensure that further applications of $\exists$-rule are not applied to the blocked nodes (and therefore ensure termination).

Definition 4 *A completion tree* $\mathbf{G}$ *is called* complete *if no expansion rule can be applied.* $\mathbf{G}$ *is* clash-free *if no node contains a clash.*

A *tableaux algorithm* for checking concept satisfiability of an $\mathcal{ALC}$ concept C w.r.t. a TBox $\mathcal{T}$ builds a completion tree for C. If a complete and clash-free tree can be obtained, the algorithm returns "satisfiable"; otherwise, if it was unable to build such a tree, it returns "unsatisfiable".

Theorem 5. *(decision procedure, [21]) The tableaux algorithm always terminates for a given $\mathcal{ALC}$ concept C and TBox $\mathcal{T}$, and returns "satisfiable" iff C is satisfiable w.r.t. a TBox $\mathcal{T}$.*

2.2 Model Checking

Definition 6 (Kripke Structure) *Let V be a set of Boolean variables. A* Kripke structure *M over V is a quadruple $M = (S, I, R, L)$ where*

1. *S is a finite set of states.*
2. *$I \subseteq S$ is the set of initial states.*
3. *$R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.*
4. *$L : S \rightarrow 2^V$ is a function that labels each state with the set of variables true in that state.*

We view each state s as a truth assignment to the variables in V. We view a set of states as a Boolean function over V, characterizing the set. For example, the set of initial states, I, is considered as a Boolean function over V. Thus, if a state s belongs to I, we write $s \models I$. Similarly, if $v_i \in L(s)$ we write $s \models v_i$, and if $v_i \notin L(s)$ we write $s \models \neg v_i$.

In practice, the full Kripke structure of a system is not explicitly given. Rather, a model is described by a set of Boolean variables $V = \{v_1, ..., v_n\}$, their initial values and their next-state assignments. The definition we give below is an abstraction of the input language of *SMV* [17].

Definition 7 (Model Description) *Let $V = \{v_1, ..., v_n\}$ be a set of Boolean variables. A tuple $MD = (I_{MD}, [\langle c_1, c_1' \rangle, ..., \langle c_n, c_n' \rangle])$ is a* Model Description *over V where I_{MD}, c_i, c_i' are Boolean expressions over V.*

The semantics of a model description defines a Kripke structure $M_{MD} = (S, I_M, R, L)$, where $S = 2^V$, $L(s) = s$, $I_M = \{s | s \models I_{MD}\}$, and $R = \{(s, s') : \forall 1 \leq i \leq n,\ s \models c_i$ implies $s' \models \neg v_i$ and $s \models c_i' \wedge \neg c_i$ implies $s' \models v_i\}$.
Intuitively, a pair $\langle c_i, c_i' \rangle$ defines the next-state assignment of variable v_i in terms of the current values of $\{v_1, ..., v_n\}$. That is,

$$\text{next}(v_i) = \begin{cases} 0 & \text{if } c_i \\ 1 & \text{if } c_i' \wedge \neg c_i \\ \{0, 1\} & \text{otherwise} \end{cases}$$

where the assignment $\{0, 1\}$ indicates that for every possible next-state value of variables $v_1, ...v_{i-1}, v_{i+1}, ..., v_n$ there must exist a next-state with $v_i = 1$, and a next-state with $v_i = 0$.

Computation Tree Logic (CTL) [12]. Given a finite set AP of atomic propositions, formulas of CTL are recursively defined as follows:

– Every atomic proposition is a CTL formula.
– If φ and ψ are CTL formulas then so are: • $\neg\varphi$ • $\varphi \wedge \psi$ • $\mathsf{AX}\varphi$ • $\mathsf{EX}\varphi$ • $\mathsf{A}[\varphi \mathsf{U} \psi]$ • $\mathsf{E}[\varphi \mathsf{U} \psi]$

Additional operators are defined as syntactic sugaring of those above:
• $\mathsf{AF}\varphi = \mathsf{A}[true\ \mathsf{U}\varphi]$ • $\mathsf{EF}\varphi = \mathsf{E}[true\ \mathsf{U}\varphi]$
• $\mathsf{AG}\varphi = \neg\mathsf{E}[true\ \mathsf{U}\neg\varphi]$ • $\mathsf{EG}\varphi = \neg\mathsf{A}[true\ \mathsf{U}\neg\varphi]$

The formal semantics of a CTL formula are defined with respect to a Kripke structure $M = (S, I, R, L)$ over a set of variables $V = \{v_1, ..., v_k\}$. A path in M is an infinite sequence of states $(s_0, s_1, ...)$ such that each successive pair of states (s_i, s_{i+1}) is an element of R. The notation $M, s \models \varphi$, means that the formula φ is true in state s of the model M.

– $M, s \models p$ iff $s \models p$
– $M, s \models \neg\varphi$ iff $M, s \not\models \varphi$
– $M, s \models \varphi \wedge \psi$ iff $M, s \models \varphi$ and $M, s \models \psi$
– $M, s_0 \models$ AXp iff for all paths $(s_0, s_1, ...)$, $M, s_1 \models p$
– $M, s_0 \models$ EXp iff there exists a path $(s_0, s_1, ...)$, $M, s_1 \models p$
– $M, s_0 \models \mathsf{A}[\varphi \mathsf{U} \psi]$ iff for all paths $(s_0, s_1, ...)$, there exists i such that $M, s_i \models \psi$ and for all $0 \leq j < i$,$M, s_j \models \varphi$
– $M, s_0 \models \mathsf{E}[\varphi \mathsf{U} \psi]$ iff there exists a path $(s_0, s_1, ...)$, and there exists i such that $M, s_i \models \psi$ and for all $0 \leq j < i$,$M, s_j \models \varphi$

We say that a Kripke structure $M = (S, I, R, L)$ satisfies a CTL formula φ ($M \models \varphi$) if there exists a state s_i such that $s_i \models I$ and $M, s_i \models \varphi$.

Linear Temporal Logic (LTL) [19]. Linear temporal logic uses the same temporal operators as CTL, but has no path quantifiers. An LTL formula is thus evaluated with respect to a given path rather than a Kripke structure. A formula φ is said to hold in a Kripke structure M if it holds along all paths that start from an initial state of M.

Fairness constraints and LTL model checking. Different definitions of fairness constraints exist in the literature [13]. The fairness definition used for model checking can be presented as the LTL formula GFp, describing a fair path as one on which the proposition p occurs infinitely often (or at least once in a loop).

Model checking of an LTL formula φ is commonly done by first building a Büchi automaton $A_{\neg\varphi}$ that accepts $\neg\varphi$ [23]. The composition of $A_{\neg\varphi}$ (presented as a state-machine) with the model M (denoted $A_{\neg\varphi}||M$) should then be empty: a path, if found, satisfies $\neg\varphi$, and therefore demonstrate a counterexample for φ. Note that the acceptance condition of a Büchi automaton requires that an accepting state is visited infinitely often. Translated into model checking notation, the formula F(false) is verified on $A_{\neg\varphi}||M$, with fairness constraints that are the Büchi acceptance conditions. Thus model checking of any LTL formula is reduced to model checking of a simple Fp formula on fair paths.

Note that the LTL formula Fp and the CTL formula AFp are equivalent. We sometimes use the CTL notation, since the description of an erroneous path, (EG¬p) is not possible in LTL.

2.3 Model description as a DL terminology

We show how a model description can be encoded as a TBox over the Description Logic dialect $\mathcal{ALC}$. This translation is taken from [3], where it was used for bounded model checking of safety formulas. In Section 3 we demonstrate how unbounded model checking of liveness formulas can be achieved.

Let $MD = (I, [\langle c_1, c'_1\rangle, ..., \langle c_n, c'_n\rangle])$ be a model description for the model $M_{MD} = (S, I, R, L)$, over $V = \{v_1, ..., v_n\}$. We generate a TBox $\mathcal{T}_{MD}$, linear in the size of MD. For each variable $v_i \in V$ we introduce one primitive concept V_i, where V_i denotes $v_i = 1$ and $\neg\mathsf{V}_i$ denotes $v_i = 0$. We introduce one primitive role R corresponding to the transition relation of the model. Given a Boolean expression p over the state variables $v_1, ..., v_n$, we denote $\mathcal{D}(p)$ the concept P derived from p by replacing each v_i in p with V_i, and $\vee, \wedge, \neg$ with $\sqcap, \sqcup, \neg$ respectively. For example, if p = $(\neg v_1 \wedge v_2)$, then $\mathcal{D}(p) = (\neg\mathsf{V}_1 \sqcap \mathsf{V}_2)$.

We define the concept S_0 to represent the set of initial states: $\mathsf{S}_0 = \mathcal{D}(I)$. We define $\mathsf{C}_i = \mathcal{D}(c_i)$, $\mathsf{C}'_i = \mathcal{D}(c'_i)$, for all $1 \leq i \leq n$. We then introduce concept inclusions describing the model: for each pair $\langle c_i, c'_i\rangle$ we introduce the inclusions

$$\begin{aligned} \mathsf{C}_i &\sqsubseteq \forall\mathsf{R}.\neg\mathsf{V}_i \\ (\neg\mathsf{C}_i \sqcap \mathsf{C}'_i) &\sqsubseteq \forall\mathsf{R}.\mathsf{V}_i \end{aligned}$$

The first inclusion ensures that in any interpretation, an individual that belongs to C_i can be related by R only to individuals that do not belong to V_i. As we show in the sequel, individuals correspond to states in the model M_{MD}. This means that when c_i holds in a state s, all neighbor states of s must have $v_i = 0$. The above inclusions thus restrict the role R to agree with the definition of R in the model description.

The TBox built above describes the model only, and does not consider the specification to be verified. Legal interpretations include for example the empty interpretation, and are not necessarily useful for verification. In order to use DL reasoning for model checking we need to add axioms to the terminology, to stand for the specification. The method we describe below adds concept inclusions that describe an *error* in the model. Interpretations will therefore be legal sub-models that demonstrate an erroneous behavior.

3 Model Checking Liveness Formulas using DL

We first consider a liveness formula of the type $\mathsf{AF}(p)$, with p being a Boolean expression. For our method to work, we need to define a *buggy* path, that is, a path on which p never happens. We thus look for a representation of $\mathsf{EG}(\neg p)$.

The following is a known equation [12], that we use for our translation into DL:

$$\mathsf{EG}(\neg p) = \neg p \wedge \mathsf{EX}(\mathsf{EG}(\neg p)) \tag{1}$$

Let $MD = (I, [\langle c_1, c'_1\rangle, ..., \langle c_n, c'_n\rangle])$ be a model description for the model $M_{MD} = (S, I, R, L)$ over $V = \{v_1, ..., v_n\}$, and let $\mathcal{T}_{MD}$ be the terminology built for it as described in Section 2.3. Let $\varphi = \mathsf{AF}(p)$ be the formula to be verified, with p being a Boolean expression over the variables $v_1, ..., v_n$. Let P= $\mathcal{D}(p)$ the corresponding concept. We introduce a new concept called $\mathtt{EGnotP}$, and add the following concept inclusion to $\mathcal{T}_{MD}$:

$$\mathtt{EGnotP} \sqsubseteq \neg\mathsf{P} \sqcap \exists\mathsf{R}.\mathtt{EGnotP} \tag{2}$$

Note that the expression $\exists\mathsf{R}.\mathsf{C}$ can be seen as taking one step through R, and thus corresponds, in a sense, to the CTL expression $\mathsf{EX}(\mathsf{C})$.

Let $\mathcal{T}'_{MD}$ be the terminology we get by adding Equation (2) to $\mathcal{T}_{MD}$. We define the concept C_φ by $\mathsf{C}_\varphi \sqsubseteq \mathsf{S}_0 \sqcap \mathtt{EGnotP}$. In order to verify φ, we now check whether C_φ is satisfiable with respect to our terminology: $\mathcal{T}'_{MD} \models_{dl} \mathsf{C}_\varphi$? A positive answer from the DL reasoning tool will be accompanied by an interpretation for $\mathcal{T}'_{MD}$ in which C_φ is not empty. This interpretation can serve as a witness to $\mathsf{EG}(\neg p)$, or as a counterexample to $\mathsf{AF}(p)$. The following proposition states our result formally.

Proposition 8. *$M_{MD} \not\models \varphi$ if and only if $\mathcal{T}'_{MD} \models_{dl} \mathsf{C}_\varphi$.*

The proof can be found in [4].

Example. Consider a model of a buggy three-bit counter, for which the least and most significant bits behave as expected, but the middle bit has a bug: when its current value is 0, it may assume any value in the next state, and when its current value is 1 it keeps its value in the next state. This behavior can be described as a model description (using

$$\begin{aligned}
\mathsf{S}_0 &\sqsubseteq (\neg\mathsf{V}_1 \sqcap \neg\mathsf{V}_2 \sqcap \neg\mathsf{V}_3)\\
\mathsf{V}_1 &\sqsubseteq \forall\mathsf{R}.\neg\mathsf{V}_1\\
\neg\mathsf{V}_1 &\sqsubseteq \forall\mathsf{R}.\mathsf{V}_1\\
\mathsf{V}_2 &\sqsubseteq \forall\mathsf{R}.\mathsf{V}_2\\
(\mathsf{V}_1 \sqcap \mathsf{V}_2 \sqcap \mathsf{V}_3) &\sqsubseteq \forall\mathsf{R}.\neg\mathsf{V}_3\\
(\neg\mathsf{V}_3 \sqcap (\neg\mathsf{V}_1 \sqcup \neg\mathsf{V}_2)) &\sqsubseteq \forall\mathsf{R}.\neg\mathsf{V}_3\\
(\mathsf{V}_1 \sqcap \mathsf{V}_2 \sqcap \neg\mathsf{V}_3) &\sqsubseteq \forall\mathsf{R}.\mathsf{V}_3\\
(\mathsf{V}_3 \sqcap (\neg\mathsf{V}_1 \sqcup \neg\mathsf{V}_2)) &\sqsubseteq \forall\mathsf{R}.\mathsf{V}_3
\end{aligned}$$

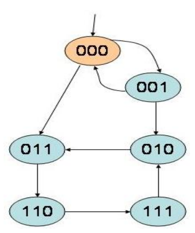

Fig. 2. Terminology and Kirpke Structure for "Counter"

$\top$ for $v_1 \vee \neg v_1$ and $\bot$ for $v_1 \wedge \neg v_1$) in the following way.
$\mathtt{Counter} = (I, [\langle v_1, \top\rangle, \langle \bot, v_2\rangle, \langle (v_1 \wedge v_2 \wedge v_3) \vee (\neg v_3 \wedge (\neg v_1 \vee \neg v_2)), \top\rangle])$ with $I = \neg v_1 \wedge \neg v_2 \wedge \neg v_3$. Figure 2 describes the Kripke structure for $\mathtt{Counter}$. Note that in the figure, v_1 is the right-most bit.

The description of the model as a TBox $\mathcal{T}_{\mathtt{Counter}}$ over $\mathcal{ALC}$ has three concepts $\mathsf{V}_3, \mathsf{V}_2, \mathsf{V}_1$ and one role R. The concept inclusions for $\mathcal{T}_{\mathtt{Counter}}$ are given in Figure 2. For convenience we broke the concept inclusions describing the behavior of V_3 into two parts. Note that there is only one concept inclusion describing the behavior of V_2, since it is free to change when its value is 0.

Let the formula to be verified be $\varphi = \mathsf{AF}(v_1 \wedge \neg v_2 \wedge v_3)$, asserting that the state (101) should be reachable on every path. Translated into DL, we add the following inclusion to $\mathcal{T}_{Counter}$ (presented in Fig. 2):

$$\texttt{EGnotP} \sqsubseteq (\neg \mathsf{V}_1 \sqcup \mathsf{V}_2 \sqcup \neg \mathsf{V}_3) \sqcap \exists \mathsf{R}.\texttt{EGnotP}$$

and define $\mathsf{C}_\varphi \sqsubseteq \mathsf{S}_0 \sqcap \texttt{EGnotP}$. Note that since the formula does not hold in the model, running the DL query $\mathcal{T}_{Counter} \models_{dl} \mathsf{C}_\varphi$ is expected to be satisfiable. A possible model (or counterexample to the formula) could be the loop (000),(001),(000)...

As discussed in Section 2.2, liveness formulas are usually accompanied by one or more fairness constraints, and need to be verified on fair paths only. In our example, let the fairness constraint be $\texttt{Fairness}\ (v_1 \wedge v_2 \wedge v_3)$, asserting that only paths on which the state (111) occurs infinitely often should be considered. The loop (000),(001),(000) is not a fair counterexample, and a different path should be sought . A fair counterexample will then be (000),(011),(110),(111),(010),(011). In the section below we discuss how fairness can be implemented in DL.

4 Realizing Fairness in Tableaux Reasoning

While a model and a liveness formula can be encoded as a terminology over $\mathcal{ALC}$, this is not the case for a fairness proposition. If, as in our case, the proposition is mapped to a primitive concept FC and we are using tableaux reasoning, then no bound is known a-priori on the depth in which the concept FC may appear in a completion tree. Thus expressing the existence of a fairness condition can be seen as *reachability*, which can not be expressed in first order logic.

We propose a modification to the tableaux procedure to support fairness. Our procedure is both terminating and sound: if a fair cycle is found, it is a correct one. However, the procedure is not complete, that is, there are cases where a fair cycle exits, but our procedure fails to find it. We show that by an iterative application of the algorithm, completeness can also be achieved. In the remainder of this section we discuss the theoretical and implementation considerations for realizing fairness in DL reasoning.

Recall from Section 2.2 that fairness constraints in model checking are variables that should be satisfied over all cycles in the model. In tableaux reasoning, a model is represented by a completion tree, and cycles in the model are represented by blocked nodes. If node x is blocked by the node x_0 then there exists a path of nodes $x_0, \ldots, x_n$ such that $\mathcal{L}(\langle x_0, x_1 \rangle) = R_0, \ldots, \mathcal{L}(\langle x_n, x \rangle) = R_n$ and R_i are the roles occurring in a terminology. This represents the blocking loop $(x_0, \ldots, x_n)^*$.

In order to implement reasoning with fairness, we need to reject those completion trees that corresponds to unfair computations. Let FC be a fairness constraint. Completion tree $\mathbf{G}$ is *unfair* w.r.t. FC if there is loop $(x_0, \ldots, x_n)^*$ such that $\mathsf{FC} \notin \mathcal{L}(x_i)$ for all $0 \leq i \leq n$. $\mathbf{G}$ is called *fair model* of a concept C w.r.t. fairness constraint FC if $\mathbf{G}$ is a model of C which is not unfair w.r.t. FC.

Modifying Tableaux to Support Fairness Our approach to implementing fairness is to build a complete and clash-free completion tree and, if it is unfair, to attempt to make it fair by adding the fairness constraint to the label of some node involved in a cycle. To accomplish this, the tableaux algorithm is extended with the new rule illustrated in Figure 3. (Note that this new rule must also have a lower priority than all existing rules.)

$fairness$-rule: if 1. x is a node blocked by x_0, $(x_0, \ldots, x_n)^*$ is a cycle corresponding to x,
2. FC is a fairness constraint such that for every $i, 0 \leq i \leq n, \mathsf{FC} \notin \mathcal{L}(x_i)$
then set $\mathcal{L}(x_i) = \mathcal{L}(x_i) \cup \{\mathsf{FC}\}$ for some $i : 0 \leq i \leq n$

Fig. 3. Expansion rule for fairness

It might be the case that there is no fair model. For example, the concept C is satisfiable w.r.t. TBox $\mathcal{T}_1 = \{C \sqsubseteq \exists R.C \sqcap \neg B\}$ without any fairness constraint but not satisfiable w.r.t. fairness constraint $\mathsf{FC} = B$. Indeed, every node of the completion tree for C will be labeled with $\neg B$, and it is not possible to add B to a cycle without a resulting clash.

Theorem 9. *The tableaux algorithm with $fairness$-rule terminates and is sound (if a complete clash-free fair completion tree for C is found then C is satisfiable).*

Proof Outline. Soundness is straightforward. To prove termination, assume WLOG that the completion tree is a single path. There are three cases to consider after a first application of $fairness$-rule to a given blocking loop: (1) it is subsequently possible to compute a complete clash-free fair completion tree before a second application of $fairness$-rule, (2) a clash occurs before a second application of the $fairness$-rule, or (3) a subsequent application of $fairness$-rule is required. Both cases (1) and (2) lead to termination. Case (3) implies that the addition of an FC to a label inside the cycle breaks the blocking condition and leads to a new cycle. The algorithm therefore proceeds by adding an FC inside the next loop. Again, there are three possible outcomes, with two resulting in termination. Ultimately, there is a sequence of case (3) that transpire for which adding an FC forces unblocking the last node and moving the blocking loop forward. However, after a finite number of occurrences of case (3), there must eventually be two nodes labeled by the same FC for which the labels are the same (since the TBox is finite). One of these nodes will then block the other, and the fair loop must then be established. □

Note that there is no guarantee of completeness, that is, if a concept is satisfiable w.r.t. FC, that the tableaux procedure builds a complete clash-free fair completion tree. This is a consequence of the way blocking is defined. To illustrate, consider a TBox consisting of two GCIs: $\mathcal{T}_2 = \{C \sqsubseteq \neg B, \top \sqsubseteq \exists R.\top\}$. Concept C is satisfiable w.r.t. $\mathcal{T}_2$ since there exists a complete and clash-free completion tree $\mathbf{G} = (V, E, \mathcal{L})$ such that $V = \{x, y\}, E = \{\langle x, y\rangle\}, \mathcal{L}(x) = \{C, \neg B, \exists R.\top\}, \mathcal{L}(y) = \{\exists R.\top\}, \mathcal{L}(\langle x, y\rangle) = R$. Here, node y is blocked by x. However, the fair algorithm with $\mathsf{FC} = B$ will return "unsatisfiable" since the clash appears immediately following the addition of FC to the only possible node x. However, there is complete and clash-free fair completion tree for this case: $\mathbf{G}' = (V', E', \mathcal{L}')$ with $V' = \{x, y, z\}, E' = \{\langle x, y\rangle, \langle y, z\rangle\}, \mathcal{L}'(x) = \{C, \neg B, \exists R.\top\}, \mathcal{L}'(y) = \{B, \exists R.\top\}, \mathcal{L}'(z) = \{\exists R.\top\}, \mathcal{L}'(\langle x, y\rangle) = \mathcal{L}'(\langle y, z\rangle) = R$. Here, node z can be label blocked by either x or y. To address this problem, we introduce the notion of n-blocking.

Definition 10 (n-Blocking) *Let n be a non-negative integer. Node x is n-*blocked *by node x_0 with blocking loop $x_0, \ldots, x_m$ if x is blocked by x_0 by the same blocking loop and $n \leq m$, that is, there are at least n nodes in the blocking loop.*

Observe that normal blocking can be viewed as a 0-blocking. Also note that replacing normal blocking with n-blocking in the (fair) tableaux algorithm will clearly preserve termination, soundness and (in the unfair case) completeness.

In the example above the algorithm with 1-blocking will find that C is satisfiable w.r.t. $\mathcal{T}_2$, producing the completion tree $\mathbf{G}'$, with node z being blocked by x.

Theorem 11. *Let C denote a concept, $\mathcal{T}$ a TBox,* FC *a fairness condition and n a non-negative integer. Then there is a tableaux-based decision procedure that returns "satisfiable" iff C is satisfiable w.r.t. $\mathcal{T}$ and* FC *with a fair blocking loop with length not exceeding n.*

Proof Outline. Check the unfair satisfiability of C using tableaux. If it is unsatisfiable, return "unsatisfiable". Then, for each $0 \leq k \leq n$, run the fair algorithm with k-blocking. Return "satisfiable" if the algorithm returns "satisfiable" for some such k; otherwise return "unsatisfiable". Termination and soundness are a simple consequence of Theorem 9. Completeness follows from the fact that no fair blocking loops for any possible length not exceeding n were found. □

This approach can be used for detecting fair cycles in model checking. The length of a cycle cannot exceed the number of states in the model. Since models are finite, for every model M and specification φ there must exist n such that if $M \models \varphi$ then M contains a fair cycle with length not exceeding n. Thus, it is possible to build the TBox $\mathcal{T}$ using the technique from Section 3 and run the procedure suggested in our proof outline for Theorem 11 to get a decision procedure for fair model checking.

5 Experimental Evaluation

We implemented the modified tableaux reasoning procedure described in Section 4 on top of FaCT++ [22], a state-of-the-art description logic reasoner. In order to run real examples, we wrote a translator from the AIGER [5] format, that builds a terminology as described in Section 3. Liveness formulas were translated in the AIGER models into Büchi automata (see section 2.2), and the fairness constraints were passed to FaCT++ using a new construct in the interface language.

The models we acquired were originally written in the VIS [9] input language, and were translated into AIGER using different tools. We present results running three sets of benchmarks with fairness constraints. The "amba" benchmark encodes an Advanced High Performance Bus. The "vsa" benchmarks encode a simple architecture for a microprocessor. In each of the vsa benchmarks, the number indicates the datawidth of the microprocessor. The "Vending" example is part of the VIS distribution.

Experimental Results. Figure 4 summarizes our proof-of-concept results. The examples were run on Intel CoreDuo computer, 3.4 HGz, 2Gb RAM, Linux SuSE 11.0. Times reported are in seconds and a time of DNF indicates that the run did not finish in the alloted time of 1 hour.

It is evident from Figure 4 that our approach is efficacious in certain scenarios. For the "amba" benchmark, our system could not finish in the given time, while VIS was easily able to handle it in a fraction of a second. However, the "vsaR" benchmarks

Benchmark	Result	Size (vars)	FaCT++	VIS
vsaR - 6	Fail	170	9.9s	DNF
vsaR - 8	Fail	204	12.3s	DNF
vending	Pass	64	DNF	1.1s
amba2 - G3	Pass	63	DNF	0.7s
amba3 - G3	Pass	77	DNF	17.7s

Fig. 4. RUN TIMES FOR THE FAIRNESS VERIFICATION TASKS.

proved simple for our reasoner while VIS was unable to finish in the given time. It seems that our method works better when a fair cycle does exist in the model. This can be explained by the fact that when a clash is found applying the fairness-rule, the n-blocking algorithm should be applied again with increased n.

6 Conclusion

We have proposed a novel approach to fair cycle detection in model checking, using tableaux-based DL technology. While encoding of fairness constraints can not be expressed as a terminology over $\mathcal{ALC}$, we showed how the tableaux reasoning procedure can be modified to support it.

Experiments, comparing our method to the model checker VIS [9], show mixed results. On some models our method significantly outperform VIS, while other models demonstrate the opposite. This is not too surprising. In the model checking community it has been recognized that no single method can outperform others on all models [18]. State of the art model checkers invoke multiple algorithms for each model checking problem, to speed up verification. Our method can fit nicely in such a platform, speeding up verification time for part of the models.

Acknowledgements

We thank Armin Biere and Barbara Jobstmann for providing us the AIGER examples. Our work was partially supported by the SEALIFE project (IST-2006-027269), by NSERC of Canada and by the Cheriton Scholarship Fund of the Cheriton School of Computer Science.

References

1. M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider. *Distributed systems: methods and tools for specification. An advanced course*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
2. Mohammad Awedh and Fabio Somenzi. Proving more properties with bounded model checking. In *CAV*, pages 96–108, 2004.
3. S. Ben-David, R. Trefler, and G. Weddell. Bounded model checking with description logic reasoning. In *Automated Reasoning with Analytic Tableaux and Related Methods*, LNAI 4548-0060, pages 60–72, July 2007.

4. Shoham Ben-David, Richard Trefler, Dmitry Tsarkov, and Grant Weddell. Checking inevitability and invariance using description logic technology, 2008. Technical report CS-2008-28, University of Waterloo.
5. A. Biere. The AIGER And-Inverter Graph (AIG) Format, 2007. http://fmv.jku.at/aiger/.
6. A. Biere, A. Cimatti, E. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, 1999.
7. Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In *Formal Methods in Computer Aided Design*, pages 37–54. Springer-Verlag, 2000.
8. Roderick Bloem, Harold N. Gabow, and Fabio Somenzi. An algorithm for strongly connected component analysis in log symbolic steps. *Formal Methods in System Design*, 28(1):37–56, 2006.
9. R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *CAV*, 1996.
10. D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning in expressive description logics with fixpoints based on automata on infinite trees. In *IJCAI*, pages 84–89, 1999.
11. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.
12. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logics of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
13. Nissim Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
14. Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Beyond safety: customized sat-based model checking. In *DAC*, pages 738–743, 2005.
15. Giuseppe De Giacomo and Fabio Massacci. Combining deduction and model checking into tableaux and algorithms for converse-PDL. *Information and Computation*, 162(1-2):117–137, 2000.
16. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE–3(2):125–143, 1977.
17. K. McMillan. Symbolic model checking, 1993.
18. Ziv Nevo. User-friendly model checking: Automatically configuring algorithms with rule-base/pe. In 4th *Haifa Verification Conference*, October 2008.
19. Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.
20. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming*, 1982.
21. Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.
22. Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.
23. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, Berlin, 1996.