Translating LTL to the Equational μ-Calculus Using Büchi Automata Optimisations

Tim Kemp

Supervisors: Jaco van de Pol, Bodo Manthey

February 1, 2018

Abstract

Two important temporal logics used in model checking are Linear Temporal Logic (LTL) and μ-calculus. LTL can be translated to μ-calculus by using Büchi automata as an intermediate form. This paper focuses on the translation from Büchi automata to the equational μ-calculus. The model checking library Spot can be used to translate LTL to Büchi automata. The optimisations performed by Spot combined with a translation from Büchi automata to an equational variant of the μ-calculus lead to an efficient translation. Properties for a system of traffic lights are modelled to illustrate the translation. The translation from LTL to μ-calculus is compared to the translation in the model checker LTSmin. This comparison shows that the translation via Büchi automata produces significantly smaller μ-calculus formulas in most cases.

1 Introduction

Model checking Model checking is used as an automated technique to verify properties of software or hardware systems. Model checking is mainly used in safety-critical systems. Safety-critical systems are those systems whose failure could result in loss of life, significant property damage or damage to the environment. When applying model checking we can distinguish three phases. These are the modelling phase, the running phase, and the analysis phase [1].

In the modelling phase the system is typically modelled as a finite automaton. This automaton consists of a finite set of states and a set of transitions between these states. A state contains information such as the current values of variables or the previously executed statement. The properties which need to be verified should also be described in some specification language. Usually a property is given as a formula in some temporal logic. A temporal logic is a logic which contains modal operators which refer to time, besides the logical operators that are used in propositional logic. Using temporal logic one can describe system properties such as safety and liveness. Safety properties state that something bad will never happen and liveness properties assert that good things will eventually happen.

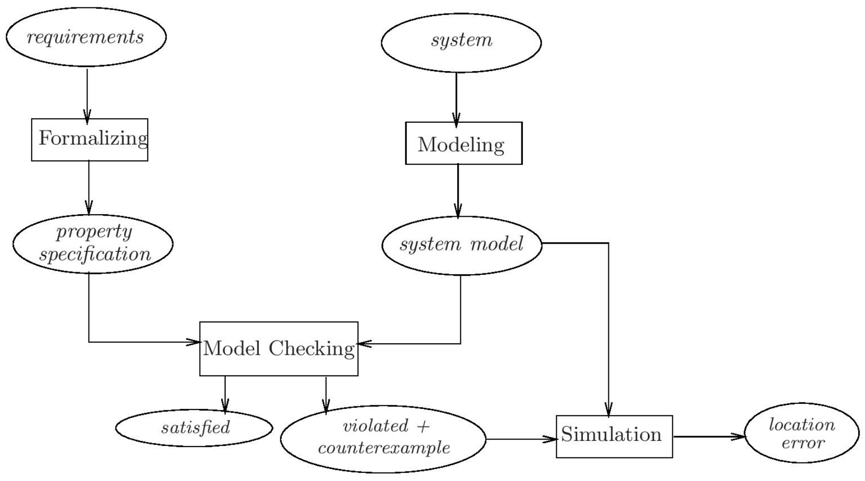

Figure 1: Schematic view of the model checking approach. Source: [1]

Once the system and the properties are specified in a language accepted by a model checker, one can run the model checker to validate a property. A model checker checks every state in a systematic way to see if the property holds. If the property holds, one can continue with the next property until all properties are checked. If the property does not hold, a model checker can provide a counterexample such that the location of the error can be found. It is also possible that there is a modelling error, where either the model or the property has to be refined. The approach to model checking is shown in Figure 1. Another possibility is that the model is too large to be analysed in reasonable time. This depends on the expressiveness of the logic that is used for model checking.

Temporal logic Two important temporal logics are Linear Temporal Logic (LTL) and μ-calculus. LTL is less expressive, but concise and admits fast model checking algorithms for concrete automata. μ-Calculus is very expressive, but all known model checking procedures take super-polynomial time. On the other hand, μ-calculus allows efficient symbolic algorithms that work on logic descriptions of automata, rather than the automata themselves.

When using a μ-calculus model checker it is convenient to still be able to specify the properties in LTL, as specifying system properties in LTL is simpler [11]. LTL formulas can be translated to μ-calculus in multiple ways. One way is by using Büchi automata as an intermediate form. Büchi automata are automata that accept infinite words as input. A recent tool to generate Büchi automata from LTL is Spot [9]. Spot applies various optimisations to achieve small automata and to reduce the number of non-deterministic states.

Motivation The current translation in the LTSmin model checker [10] is based on a translation using tableaux rules [6]. Due to the optimisations performed by Spot, improvement can be made by using Büchi automata generated by Spot instead. It is also possible to improve the translation by using the equational

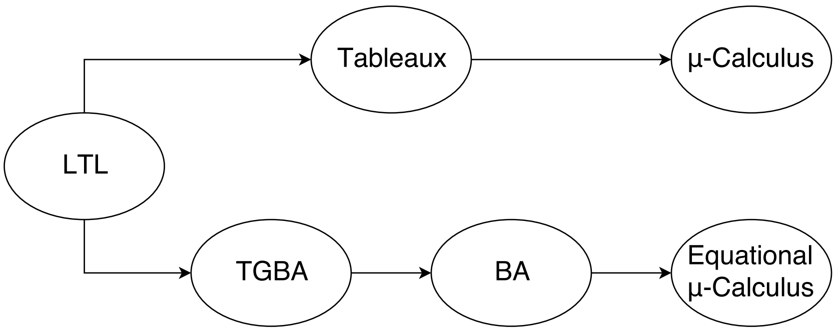

Figure 2: Schematic view of the two discussed translations from LTL to μ-calculus

μ-calculus, which is more concise than the regular μ-calculus. This paper introduces a translation from Büchi automata to the equational μ-calculus. A schematic view of this translation compared to the translation used in LTSmin can be seen in Figure 2. To illustrate the use of model checking and to analyse the translation, a list of system properties is modelled for a system of traffic lights. These properties are formalised in LTL. The results of the translation are compared to the translation in the LTSmin model checker.

Organisation The organisation of this paper is as follows. Below some related work is given. In Section 2 the temporal logic LTL, Büchi automata, and μ-calculus are defined and explained. In the following section it is shown how to translate LTL to the equational μ-calculus. In the section after that properties of a system of traffic lights are modelled. Section 5 analyses the efficiency of the translation and the final section contains the conclusions and possibilities for future research.

Related work Model checking and temporal logic are explained in the book "Principles of model checking" [1]. An overview and description of Spot and model checking in general, including the translation from LTL to transition-based generalised Büchi automata (TGBA), is given in "Contributions to LTL and ω-Automata for Model Checking" [7]. The translation from LTL to TGBA and the degeneralisation from TGBA to Büchi automata performed by Spot is explained in more detail in "LTL translation improvements in Spot 1.0" [8]. This degeneralisation will be analysed in section 3.1. The algorithm that is used by Spot is originally introduced and proven to be correct in the paper "On-the-Fly Verification of Linear Temporal Logic" [5].

A translation from CTL*, which is a superset of LTL, to μ-calculus was first defined by Mads Dam in the paper "CTL* and ECTL* as Fragments of the Modal mu-Calculus" [6]. A more efficient translation has been given by Bhat, Cleaveland, and Groce in the paper "Efficient Model Checking Via Büchi Tableau Automata" [2].

Other sources that I have used to understand μ-calculus are "The mu-calculus and model-checking" [4] and "Modal Logics and mu-Calculi: An Introduction" [3].

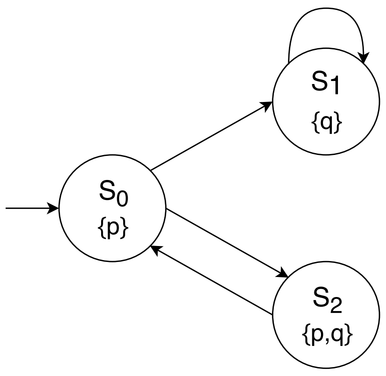

Figure 3: A Kripke structure

2 Temporal Logic

Temporal logics are logics which reason about time. These logics are used in model checking to formalise desired properties of a system. The semantic meaning of these logics can thus be defined in relation to a model of a system. Such a model is often presented as a Kripke structure.

A Kripke structure can be represented as a graph. In a Kripke structure the nodes represent the reachable states of a system, the transitions define how the system moves from one state to another and the labeling of the states maps each state to a set of propositions that hold in that state. The definition of a Kripke structure is given in Definition 1. An atomic proposition is an assertion that must be either true or false.

Definition 1 (Kripke structure)**.** Let *Prop* be a finite set of atomic propositions. A Kripke structure is a triple $\langle S, R, L\rangle$ where S is the set of states, $R \subseteq S \times S$ is a transition relation, and $L : S \rightarrow 2^{Prop}$ is a labelling of the states. A trace t of K starting in s_0 is a sequence of states $s_0, s_1, s_2, ...$ such that for all $i \geq 0, (s_i, s_{i+1}) \in R$. A word w over a trace t is a sequence of sets of propositions that hold in those states: $w = L(s_0), L(s_1), L(s_2), ...$

An example of a Kripke structure is given in Figure 3. It can be seen that once state S_1 is reached, the system will always stay in that state.

2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) is a logic which is used in model checking to describe properties that should hold in a hardware or software system. LTL formula describe the future of paths along states, for example by stating that a certain proposition will eventually be true. Since it is a linear logic, there is always a single successor moment. This is in contrast to branching temporal logic, such as Computation Tree Logic. In a branching temporal logic there are multiple paths possible in the future. The syntax of Linear Temporal Logic is given in the following definition.

Definition 2 (Syntax of LTL)**.** The set of LTL is defined inductively as follows, where AP is a set of atomic propositions.

- If $p \in AP$ then p is an LTL formula.
- If ψ and ϕ are LTL formulas then $\neg\psi, \psi \vee \phi, \mathbf{X}\psi$ (next), and $\phi\mathbf{U}\psi$ (until) are LTL formulas.

The meaning of these temporal operators is with regard to infinite sequences of states. The intuitive semantic meaning of the $\mathbf{X}$ operator applied to ψ is that ψ has to hold in the next state. The meaning of the $\mathbf{U}$ operator in $\phi\mathbf{U}\psi$, is that ϕ has to hold until ψ holds and that ψ will eventually hold. The additional logical operators $\wedge$, $\implies$, $\iff$, True and False are defined as follows.

$$\begin{aligned}
\psi \wedge \phi &:= \neg(\neg\psi \vee \neg\phi)\\
\psi \implies \phi &:= \neg\psi \vee \phi\\
\psi \iff \phi &:=(\psi \implies \phi) \wedge (\phi \implies \psi)\\
\text{True} &:= p \vee \neg p, p \in AP\\
\text{False} &:= \neg\text{True}
\end{aligned}$$

The additional temporal operators are $\mathbf{G}$ (globally), $\mathbf{F}$ (finally), and $\mathbf{R}$ (release). $\mathbf{G}\phi$ means that ϕ holds at all future states, $\mathbf{F}\phi$ means that ϕ holds at some future state, and $\psi\mathbf{R}\phi$ means that ϕ holds until and including the point where ψ becomes true (ψ does not have to become true). They can be defined using $\mathbf{U}$ as follows.

$$\begin{aligned}
\mathbf{F}\phi &:= \text{True}\mathbf{U}\phi\\
\mathbf{G}\phi &:= \neg\mathbf{F}\neg\phi\\
\psi\mathbf{R}\phi &:= \neg(\neg\phi\mathbf{U}\neg\psi)
\end{aligned}$$

Common LTL formulas are $\mathbf{GF}a$ and $\mathbf{G}(a \implies \mathbf{F}b)$. These represent two liveness properties. The former means there will be infinitely many a's and the latter means that after every a there will eventually be a b.

Whether or not an infinite sequence of states satisfies an LTL formula is defined by the semantics. The semantics are defined in Definition 3.

Definition 3 (Semantics of LTL)**.** Given a set of atomic propositions AP, a state s is a truth valuation $AP \to \{\text{True, False}\}$. It can be represented as the set of true atoms. A word $w = (s_0, s_1, s_2, ...)$ is an infinite sequence of states. Define w^i as the suffix of w starting at i: $w^i := (s_i, s_{i+1}, s_{i+2}, ...)$. The satisfaction relation $\models$ between a word and an LTL formula is defined as follows.

- $w \models p$ if $p \in s_0$
- $w \models \neg p$ if $w \not\models p$
- $w \models \psi \vee \phi$ if $w \models \psi$ or $w \models \phi$

- $w \models \mathbf{X}\psi$ if $w^1 \models \psi$
- $w \models \phi\mathbf{U}\psi$ if there exists $i \geq 0$ such that $w^i \models \psi$ and for all $0 \leq k < i, w^k \models \phi$

A word w satisfies an LTL formula ψ if $w \models \psi$.

As LTL and other temporal logics are often interpreted with respect to Kripke structures, semantics for LTL with respect to a Kripke structure is also given in Definition 4.

Definition 4 (Semantics of LTL with respect to a Kripke structure)**.** Given a Kripke structure $K = \langle S, R, L\rangle$ and a state $s \in S$, an LTL formula ϕ is satisfied ($K, s \models \phi$) if and only if for every trace t of K starting in s, the word w over this trace satisfies ϕ ($w \models \phi$).

2.2 Büchi automata

Besides using logic formulas, the specifications of model checking properties can also be described by ω-automata. These are automata that accept infinite strings as input. One type of ω-automata are Büchi automata (BA). In this paper Büchi automata are used as an intermediate form to translate LTL to μ-calculus. The definition of a Büchi automaton is given in Definition 5.

Definition 5 (BA)**.** A Büchi automaton is a tuple $\mathcal{B} = \langle AP, Q, q^0, F, \sigma\rangle$ where AP is a set of atomic propositions, Q is a finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is a set of acceptance states, $\sigma \subseteq Q \times 2^{AP} \times Q$ is a transition relation in which each transition is labelled by a Boolean assignment.

An infinite word $c_0c_1c_2... \in (2^{AP})^\omega$ of assignments is accepted by $\mathcal{B}$ if there exists a run, $(q^0, c_0, q_1)(q_1, c_1, q_2)(q_2, c_2, q_3)... \in \sigma^\omega$, that visits the acceptance set infinitely often ($\forall i \geq 0, \exists j \geq i, q_j \in F$).

Usually a Büchi automaton is represented by its visualisation instead of a tuple. The states Q are the nodes, the initial state q^0 is marked by an initial transition and the acceptance states are marked by a double circle around them. The boolean assignment of the AP is shown on the transitions.

An example of a simple Büchi automaton is given in Figure 4. This Büchi automaton accepts words that start with zero or more assignments where $a \wedge \neg b$ hold. As state 0 is the only accepting state, there should in a finite number of steps eventually be an assignment of b in the word. Afterwards there is a 1 (True) on the transition, which means that any assignment is accepted. Thus this Büchi automaton accepts only the words that are satisfied by the LTL formula

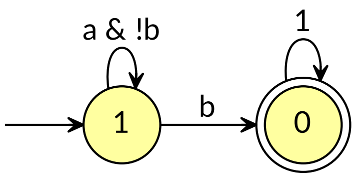

Figure 4: A BA with AP = {a,b}

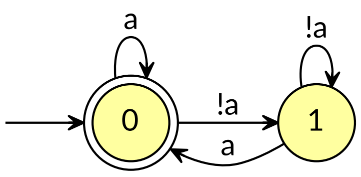

Figure 5: A BA with AP = {a}

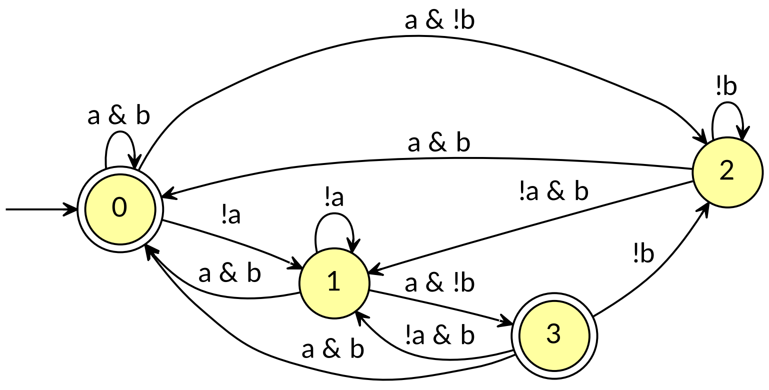

Figure 6: A Büchi automaton recognising the LTL formula $\mathbf{GF}a \wedge \mathbf{G}(a \implies \mathbf{F}b)$

$a\mathbf{U}b$. Another example is shown in Figure 5. This Büchi automaton accepts only the words that are satisfied by the LTL formula $\mathbf{GF}a$ (infinitely many a's). All transitions with label a lead towards state 0. These transitions are also the only transitions towards state 0. Since state 0 has to be visited infinitely often for a word to be accepted, this Büchi automaton accepts only the words with infinitely many a's.

As seen in Figure 4, transitions with the same start and end state are grouped and the boolean assignment is notated using propositional logic. E.g. the transitions $(1,(a,b),0)$ and $(1,(\neg a,b),0)$ are combined into $(1,b,0)$.

As a bigger example we consider the Büchi automaton $\mathcal{B} = \langle AP, Q, q^0, F, \sigma\rangle$, where

- $AP = \{a, b\}$
- $Q = \{0, 1, 2, 3\}$
- $q^0 = 0$
- $F = \{0, 3\}$
- $\sigma = \{(0, a \wedge b, 0), (0, \neg a, 1), (0, a \wedge \neg b, 2), (1, \neg a, 1), (1, a \wedge \neg b, 3), (1, a \wedge b, 0), (2, a \wedge b, 0), (2, \neg a \wedge b, 1), (2, \neg b, 2), (3, a \wedge b, 0), (3, \neg a \wedge b, 1), (3, \neg b, 2)\}$

This Büchi automaton recognises the LTL formula $\mathbf{GF}a \wedge \mathbf{G}(a \implies \mathbf{F}b)$. A visualisation is given in Figure 6.

2.2.1 Transition-based generalised Büchi automata

A variant of Büchi automata are transition-based generalised Büchi automata (TGBA). TGBA are used internally by Spot to translate LTL to Büchi automata. They are being used in this paper to analyse this translation.

Instead of having acceptance states as in a Büchi automaton, it is possible to have acceptance sets consisting of multiple states. In this case an infinite word is accepted if it visits each acceptance set infinitely often. These automata are called generalised Büchi automata (GBA). It is also possible to put the acceptance marks on the transitions instead of the states. A combination of these changes gives us transition-based generalised Büchi automata. The definition of a TGBA is given in Definition 6.

Definition 6 (TGBA)**.** A transition-based generalised Büchi automaton is a tuple $\mathcal{T} = \langle AP, Q, q^0, F, \sigma\rangle$ where AP is a set of atomic propositions, Q is a finite set of states, $q^0 \in Q$ is the initial state, $F = \{f_1, f_2, .., f_n\}$ is a finite set of acceptance marks, $\sigma \subseteq Q \times 2^{AP} \times 2^F \times Q$ is a transition relation in which each transition is labelled by a Boolean assignment and a set of acceptance marks.

An infinite word $c_0c_1c_2... \in (2^{AP})^\omega$ of assignments is accepted by $\mathcal{T}$ if there exists a run, $(q^0, c_0, F_0, q_1)(q_1, c_1, F_1, q_2)(q_2, c_2, F_2, q_3)... \in \sigma^\omega$, that visits each acceptance mark infinitely often $(\forall f \in F, \forall i \geq 0, \exists j \geq i, f \in F_j)$.

TGBA have the advantage that they are usually smaller (number of states and transitions) than BA. The visualisation of a TGBA is similar to that of a BA. Instead of having finishing states, a set of acceptance marks is shown on the transitions. A TGBA can always be degeneralised into a Büchi automaton, as will be shown in Section 3. As an example we consider the TGBA that is equivalent to the Büchi automaton in Figure 6.

This TGBA is shown in Figure 7 and is given by $\mathcal{T} = \langle AP, Q, q^0, F, \sigma\rangle$, where

- $AP = \{a, b\}$
- $Q = \{0, 1\}$
- $q^0 = 0$
- $F = \{f_0, f_1\}$
- $\sigma = \{(0, a \wedge b, \{f_0, f_1\}, 0), (0, \neg a, \{f_1\}, 0), (0, a \wedge \neg b, \{f_0\}, 1),$ $(1, a \wedge b, \{f_0, f_1\}, 0), (1, \neg a \wedge b, \{f_1\}, 0), (1, a \wedge \neg b, \{f_0\}, 1), (1, \neg a \wedge \neg b, \{\}, 1), \}$

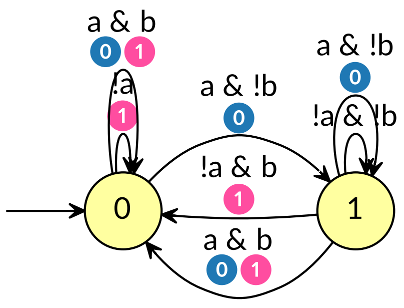

Figure 7: A TGBA recognising the LTL formula $\mathbf{GF}a \wedge \mathbf{G}(a \implies \mathbf{F}b)$

2.3 μ-Calculus

The μ-calculus is a very expressive logic which is used by model checkers where the model is usually defined as a Kripke structure (Definition 1). An equational variant of the μ-calculus is used in this paper for the translation of LTL to μ-calculus. The equational μ-calculus is described at the end of this section, after an explanation of the regular μ-calculus. The μ-calculus is a fixed point logic and uses the least fixed point operator μ and the greatest fixed point operator ν, as shown in the syntax in Definition 7. These fixed point operators allow recursive definitions.

Definition 7 (Syntax of μ-calculus)**.** Let *Var* be a finite set of variables and let *Prop* be a finite set of propositions. The set of μ-calculus formulas, F_μ is the smallest set containing:

- p for all propositions $p \in Prop$
- Z for all variables $Z \in Var$
- $\neg\phi$ if ϕ is a formula in F_μ
- $\phi \wedge \psi$ if ϕ, ψ are formulas in F_μ
- $[\cdot]\phi$ if ϕ is a formula in F_μ
- $\nu Z.\phi$ if $Z \in Var$ is a variable and ϕ is a formula in F_μ, provided that every free occurrence of ϕ in Z occurs positively, i.e. within the scope of an even number of negations.

The positivity requirement on the ν operator is to ensure that $\phi(Z)$ is a monotonic function. This is necessary for the existence of a least and greatest fixed point.

The additional syntax can be defined as follows.

$$\begin{aligned}
\phi \vee \psi &:= \neg(\neg\phi \wedge \neg\psi) \\
\langle\cdot\rangle\phi &:= \neg[\cdot]\neg\phi \\
\mu Z.\phi(Z) &:= \neg\nu Z.\neg\phi(\neg Z)
\end{aligned}$$

The meaning of $[\cdot]\phi$ is that ϕ holds after every transition. The meaning of $\langle\cdot\rangle\phi$ is that there exists an transition such that ϕ holds after this transition.

The μ operator can be interpreted as liveness and the ν operator can be interpreted as safety. The safety is seen in the examples below.

$$\begin{aligned}
&\nu Z.p \wedge [\cdot]Z \\
&\nu Z.q \vee (p \wedge [\cdot]Z)
\end{aligned}$$

The first equation means that p is true along every path. The second one means that on every path, p holds as long as q fails. With these recursive definitions, $\nu Z.\phi$ can be interpreted as looping through Z. In μ formulas the recursive looping cannot continue forever, thus eventually something should happen. This can be interpreted as finite looping. The liveness can be seen in the examples below.

$$
\begin{aligned}
&\mu Z.p \vee \langle\cdot\rangle Z \\
&\mu Z.q \vee (p \wedge \langle\cdot\rangle Z)
\end{aligned}
$$

The first one means that there exists a path such that p is eventually true after a finite number of steps. The second one means that there exists a path where p holds until q holds and q eventually holds.

It is also possible to nest μ and ν operators. Doing so increases the expressive power if the two fixed points are nested alternatively. The example below means that p is infinitely often true on some path.

$$\nu Y.\mu X.(p \wedge \langle\cdot\rangle Y) \vee \langle\cdot\rangle X \tag{1}$$

The formal meaning of μ-calculus formulas is given by the semantics in Definition 8.

Definition 8 (Semantics of μ-calculus)**.** μ-calculus formulas are interpreted with respect to Kripke structures and environments that assign meaning to propositional variables. The semantic function $[\![\cdot]\!]_i : \phi \to 2^S$ maps from basic formulas to sets of states satisfying the formula. Given a Kripke structure $K = \langle S, R, L\rangle$ and an interpretation i of the variables Z of the μ-calculus, the function is defined as follows.

- $[\![p]\!]_i = L(p)$
- $[\![Z]\!]_i = i(Z)$
- $[\![\phi \wedge \psi]\!]_i = [\![\phi]\!]_i \cap [\![\psi]\!]_i$
- $[\![\neg\phi]\!]_i = S \setminus [\![\phi]\!]_i$
- $[\![[\cdot]\phi]\!]_i = \{s \in S | \forall t \in S, (s,t) \in R \implies t \in [\![\phi]\!]_i\}$
- $[\![\nu Z.\phi]\!]_i = \bigcup\{T \subseteq S | T \subseteq [\![\phi]\!]_{i[Z:=T]}\}$, where $i[Z := T]$ maps Z to T and preserves the other mappings of i.

The satisfaction by a Kripke structure K and a state s of a formula ϕ is usually denoted as follows: $K, s \models \phi$ if $s \in [\![\phi]\!]_i$.

A variant of μ-calculus is the equational μ-calculus. It is defined in Definition 9.

Definition 9 (Equational μ-calculus)**.** An equational system E consists of a finite sequence of equations. Each equation is of the form $\nu X_i = \psi_i$ or $\mu X_i = \psi_i$. In an equational system $E = \{\lambda X_i = \psi_i\}$, where λ is either ν or μ, the X_i are distinct and the ψ_i are basic μ-calculus formulas (do not contain μ or ν). A formula in the equational μ-calculus is written as X_0 in E, where X_0 is a variable corresponding to an equation in E.

A formula X_0 in E, can be written as a regular μ-calculus formula $X_0.\phi$ by recursively substituting the variables X_i in ϕ with a formula $\lambda X_i.\psi$ corresponding to equation $\lambda X_i = \psi$. Thus, defining separate semantics for the equational μ-calculus is not necessary.

An example of the equational μ-calculus is given below. This example corresponds to Formula 1 given above, meaning that p is infinitely often true on some path.

$$\begin{aligned}\nu X_0 =& (p \wedge \langle\cdot\rangle X_0) \vee \langle\cdot\rangle X_1 \\ \mu X_1 =& (p \wedge \langle\cdot\rangle X_0) \vee \langle\cdot\rangle X_1\end{aligned}$$

3 Translations

When performing model checking using Büchi automata, the usual approach is to translate the negation of an LTL formula ϕ to a Büchi automaton $\mathcal{B}_{\neg\phi}$. The words that are accepted by this Büchi automaton violate the property ϕ and represent the forbidden behaviour. The model M is given as a Kripke structure K whose traces represent all the possible behaviour of the model. Checking whether M satisfies ϕ is done by checking the emptiness of the product of these automata: $K \otimes B_{\neg\phi}$. If the language corresponding to this product is empty, there is no forbidden behaviour in the model. When the model checker uses μ-calculus instead, the LTL formula ϕ does not need to be negated. As a μ-calculus formula is already interpreted with regard to a Kripke structure K.

To translate an LTL formula to μ-calculus, the formula is first converted to a Büchi automaton. All LTL formulas can be expressed as a Büchi automaton. A recent tool for this translation is Spot [9]. Spot translates LTL to TGBA, which can then be degeneralised into a BA. The translation from LTL to TGBA is outside the scope of this paper, but the degeneralisation of a TGBA into a Büchi automaton as done by Spot is explained below. In Section 3.2 a translation from Büchi automata to the equational μ-calculus (Definition 9) is given.

3.1 TGBA to BA

A TGBA with n states and m acceptance marks can be degeneralised by cloning the TGBA $m + 1$ times and redirecting transitions based on the acceptance marks they carry. The precise procedure is as follows [8].

If $\mathcal{T} = \langle AP, Q, q^0, F, \sigma\rangle$ is a TGBA with m acceptance conditions $F = \{f_1, ..., f_m\}$, then an equivalent Büchi automaton $\mathcal{B} = \langle AP, Q', q^{0'}, F', \sigma'\rangle$ can be constructed as follows.

- $Q' = Q \times \{0, ..., m\}$ the original automaton is cloned in $m + 1$ levels,
- $F' = Q \times \{m\}$ states from the last level are accepting,

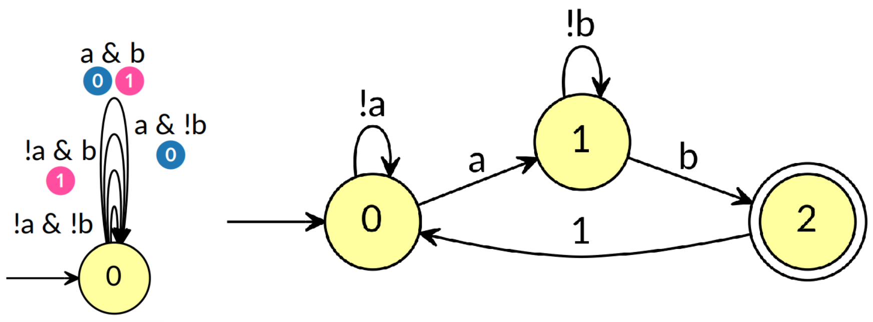

Figure 8: Degeneralisation of a TGBA into a BA without optimisations

- $\sigma' = \{((s, j), l, (d, L_j(F)))|(s, l, F, d) \in \sigma\}$ where

$$L_j(F) = \begin{cases} 0 & \text{if } j = m \\ j+1 & \text{if } j \neq m \ \& \ f_{j+1} \in F \\ j & \text{otherwise} \end{cases}$$

  for each level $j < m$ the outgoing transitions that carry f_{j+1} are redirected to the next level and all outgoing transitions from the last level are redirected to the first one.

- $q^{0'} = q^0 \times 0$ the initial state is on the first level.

An example of this degeneralisation can be found in Figure 8. On the left a TGBA recognising the LTL formula $\mathbf{GF}a \wedge \mathbf{GF}b$ is shown. On the right a Büchi automaton recognising the same formula is shown. It is obtained by copying the TGBA three times. In the first copy (state 0) the two transitions with acceptance mark 0 are redirected to the second copy. In the second copy (state 1) the two transitions with acceptance mark 1 are redirected to the third copy. In the final copy (state 2) all transitions are redirected to the first copy.

The L_j function can be optimised for transitions that carry multiple acceptance conditions, but this translation suffices to calculate the resulting size in the worst case scenario. Afterwards Spot applies multiple other optimisations to reduce the size of the Büchi automaton, but in the worst case the number of states of this automaton is $n \times (m + 1)$.

3.2 Büchi automata to the equational μ-calculus

Below a translation is introduced to translate Büchi automata to the equational μ-calculus. A Büchi automaton $\mathcal{B} = \langle AP, Q, q^0, F, \sigma\rangle$ can be translated to the equational μ-calculus as follows.

Algorithm 1

1. Introduce a corresponding μ-calculus variable X_i for each state $q_i \in Q$. Let $f : Q \to \{X_0, X_1, ..., X_n\}$ be a function that maps a state to the corresponding variable.

2. Define $S_i = \{(l, q) \mid (q_i, l, q) \in \sigma\}$ for each state $q_i \in Q$. Define S_i as the set of pairs of boolean labels and the corresponding reachable states from q_i by transitions with those labels.

3. For each state $q_i \in Q$:

   Given $S_i = \{(l_1, q_1), (l_2, q_2), ..., (l_n, q_n)\}$, generate an equation:

   $$X_i = (l_1 \wedge [\cdot]X_1) \vee (l_2 \wedge [\cdot]X_2) \vee ... \vee (l_n \wedge [\cdot]X_n)\ \forall (l_j, q_j) \in S_i$$

   where $X_0 = f(q_1), X_1 = f(q_2), ..., X_n = f(q_n)$.

4. For each equation $X_i = \phi$, set the left hand side of the equation to

   $$\begin{aligned}&\nu X_i \text{ if } X_i \text{ corresponds to an acceptance state in } F\\ &\mu X_i \text{ if } X_i \text{ does not correspond to an acceptance state in } F\end{aligned}$$

5. Order the equations such that the ν equations occur before the μ equations. Define E as the sequence of these equations.

6. The μ-calculus formula is now X_0 in E, where X_0 is $f(q^0)$.

An example of this translation is given below, which translates the Büchi automaton from Figure 9.

Example 1.

1. Introduce the μ-calculus variables X_0 and X_1 and define $f(0) = X_0$ and $f(1) = X_1$.

2. Define $S_0 = \{(\neg p \vee q, 0), (p \wedge \neg q, 1)\}$ and $S_1 = \{(q, 0), (\neg q, 1)\}$.

3. Generate equations

   $$\begin{aligned}X_0 =& ((\neg p \vee q) \wedge [\cdot]X_0) \vee (p \wedge \neg q[\cdot]X_1)\\ X_1 =& (q[\cdot]X_0) \vee (\neg q \wedge [\cdot]X_1)\end{aligned}$$

4/5. Define $E =$

   $$\begin{aligned}\{\nu X_0 =& ((\neg p \vee q) \wedge [\cdot]X_0) \vee (p \wedge \neg q[\cdot]X_1),\\ \mu X_1 =& (q[\cdot]X_0) \vee (\neg q \wedge [\cdot]X_1)\}\end{aligned}$$

6. The resulting μ-calculus formula is X_0 in E.

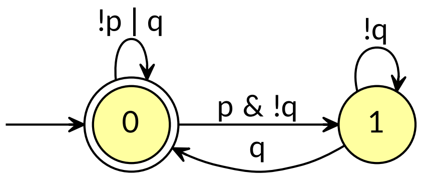

Figure 9: Büchi automaton recognising the LTL formula $\mathbf{G}\ (p \implies \mathbf{F}\ q)$

A bigger example of an equational system is shown below. This corresponds to the LTL formula $\mathbf{GF}a \wedge \mathbf{G}(a \implies \mathbf{F}b)$, the corresponding Büchi automaton is shown in Figure 6.

$$\begin{aligned}
\nu X_0 =&(a \wedge b \wedge [\cdot]X_0) \vee (\neg a \wedge [\cdot]X_1) \vee (a \wedge \neg b \wedge [\cdot]X_2)\\
\nu X_3 =&(a \wedge b \wedge [\cdot]X_0) \vee (\neg a \wedge b \wedge [\cdot]X_1) \vee (\neg b \wedge [\cdot]X_2)\\
\mu X_1 =&(a \wedge b \wedge [\cdot]X_0) \vee (\neg a \wedge [\cdot]X_1) \vee (a \wedge \neg b \wedge [\cdot]X_3)\\
\mu X_2 =&(a \wedge b \wedge [\cdot]X_0) \vee (\neg a \wedge b \wedge [\cdot]X_1) \vee (\neg b \wedge [\cdot]X_2)
\end{aligned}$$

3.2.1 Correctness

Given a Kripke structure $K = \langle S, R, L\rangle$ and an initial state $s_0 \in S$, a translation from an LTL formula ϕ to a μ-calculus formula ψ is correct if $K, s_0 \models \phi \iff K, s_0 \models \psi$ (Definition 4 and 8). The correctness of the algorithm that is used by Spot is already proven. In Claim 1 we therefore assume the construction of a correct Büchi automaton.

Claim 1. *Given a Kripke structure $K = \langle S, R, L\rangle$ and an initial state $s_0 \in S$, let $\mathcal{B}_\phi = \langle AP, Q, q^0, F, \sigma\rangle$ be a Büchi automaton that accepts exactly the infinite words over the alphabet 2^{AP} that satisfy an LTL formula ϕ. Let ψ be a μ-calculus formula generated from $\mathcal{B}_\phi$ by algorithm 1, then $K, s_0 \models \phi \iff K, s_0 \models \psi$.*

The labels of transitions in Büchi automata correspond to the labels of states in Kripke structures. Thus intuitively it can be seen that the possible runs of a Büchi automaton correspond to the possible traces of a Kripke structure described by the equational μ-calculus formula. The infinite behaviour can be intuitively seen as follows. As there can only be looped through μ a finite number of times, eventually there should be a loop through a ν variable. By step 4 of Algorithm 1 this corresponds to visiting an accepting state in the Büchi automaton.

4 Modelling traffic lights properties

Model checking is a decision procedure to check if a model satisfies a property, as explained in the introduction. In this section we shall focus on the modelling of

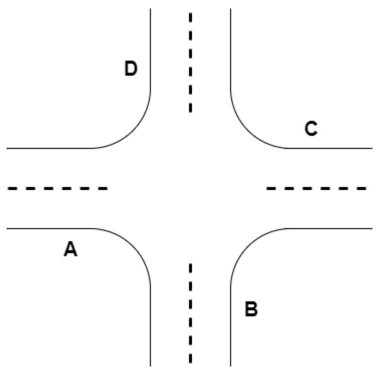

Figure 10: An overview of the four directions of crossroads with traffic lights.

such properties in LTL. Common properties in software and hardware systems are safety properties and liveness properties. Safety properties state that something bad will never happen and liveness properties assert that good things will eventually happen. One liveness property is freedom of starvation, which states that whenever a request is made, access to a resource is eventually granted. This resource can be access to a variable for example. Another liveness property is that something happens infinitely often, which shows that the program is not frozen and progress is being made.

Model checking is mainly used for safety-critical systems, as described in the introduction. An example of a safety-critical system is a system of traffic lights. The safety-critical part is that all the traffic lights should never be green at the same time. This example is chosen because some frequently occurring system properties can be specified in this model, such as the two liveness properties mentioned above.

We consider a system of traffic lights. There are a total of four traffic lights, each one having three lights: green, yellow and red. There are also sensors on the roads to detect the cars. An illustration is seen in Figure 10. For each direction a, b, c, and d there is a boolean variable for each of the three lights indicating whether the light is on. These variables are named $a_g, a_y, a_r, b_g, ...$, etc. For each sensor there is also a boolean variable indicating whether there is a car on the sensor or not: s_a, s_b, s_c, s_d. Thus in total there are 16 variables. We assume there is a Kripke structure K describing this system, such that each state in K is labelled by the set of variables that are true in that state.

The following safety properties can be specified. Property 5 is superfluous, but is added for the sake of analysis of the translation. These are properties that should always hold. The first five properties should hold for each traffic light, the last two properties describe the system in general.

1. If a green light is on, it should stay green until the yellow light goes on.
2. If a yellow light is on, it should stay yellow until the red light goes on.
3. If a red light is on, it should stay red until the green light goes on.
4. Exactly one light is on, either the green, the yellow or the red.

5. If the green light is on, the red light cannot be on in the next state.
6. If the green light is on in direction a or c, the red light should be on in direction b and d.
7. If the green light is on in direction b or d, the red light should be on in direction a and c.

The following liveness properties can be specified. Only one of these two properties need to hold, depending on whether the system uses sensors.

8. For each traffic light the green light should be on infinitely often.
9. If a car is on the road sensor, the light should eventually become green.

4.1 Properties in LTL

Using LTL it is possible to formalise the informal requirements. The nine properties are given below. All of them start with the **G** operator, as the properties should always hold.

1. $\mathbf{G}(a_g \implies (a_g \mathbf{U} a_y))$
2. $\mathbf{G}(a_y \implies (a_y \mathbf{U} a_r))$
3. $\mathbf{G}(a_r \implies (a_r \mathbf{U} a_g))$
4. $\mathbf{G}((a_r \wedge \neg a_y \wedge \neg a_g) \vee (\neg a_r \wedge a_y \wedge \neg a_g) \vee (\neg a_r \wedge \neg a_y \wedge a_g))$
5. $\mathbf{G}(a_g \implies \neg \mathbf{X} a_r)$
6. $\mathbf{G}((a_g \vee c_g) \implies (b_r \wedge d_r))$
7. $\mathbf{G}((b_g \vee d_g) \implies (a_r \wedge c_r))$
8. $\mathbf{GF} a_g$
9. $\mathbf{G}(s_a \implies \mathbf{F} a_g)$

Properties 1, 2, 3, 4, 5, 8 and 9 are also valid for direction b, c and d, so in total there are 30 properties to be checked. In the following section the results of translating these and other properties to μ-calculus are presented.

5 Experimental results

Experiment The translation has been analysed by comparing it to the translation in the LTSmin model checker [10]. The translations are compared by using the size of the resulting μ-calculus formulas as a benchmark. This size is measured as the number of μ and ν variables. This is done for both the LTL specification of the traffic lights model and a random set of LTL formulas. The former has the advantage that it contains formulas that are frequently used in model checking and the latter adds more robustness to the comparison.

The random LTL formulas that have been tested are generated using Spot. Spot contains a command line tool `randltl` to generate a list of random formulas. The propositions that are used in these formulas are given as arguments. The `-n` parameter sets the number of formulas to be generated and the `-p` parameter makes the output fully parenthesized. The likeliness of an operator to occur is determined by the priority. The probability of an operator being selected is this priority divided by the sum of the priorities of all considered operators. Four operators that are not supported by LTSmin have been disabled by setting their priorities to 0. This can be seen in the command below, which is used to generate the formulas.

```
# randltl -p -n50 a b c --ltl-priorities 'xor=0, W=0, M=0, R=0'
```

This command generates 50 different random formulas that contain at most three atomic propositions: a, b and c. The formulas contain the following operators.

$$\textbf{true}, \textbf{false}, \neg, \vee, \wedge, \implies, \iff, \mathbf{F}, \mathbf{G}, \mathbf{X}, \mathbf{U}$$

The priorities of these operators are all set to one by default, which means that every operator is equally likely to occur. Atomic propositions have a priority of three by default, thus they occur three times more likely than any given operator. We have also tried to do a comparison for longer formulas using the command below.

```
# randltl -p -n50 a b c d e f g h i j --ltl-priorities 'xor=0, W=0, M=0, R=0'
```

For each translation a shell script is written to translate a list of formulas and to count the number of μ and ν variables in the output. There is also a shell script to convert the syntax of Spot to the syntax of LTSmin. These scripts are shown in Appendix A. The translation from Algorithm 1 has been implemented in Python, but it has not been used for the results as the number of μ and ν variables is equal to the number of states of the Büchi automata.

Results The result of the comparison of the properties of the traffic lights system (Section 4.1) is seen in Table 1. Properties that result in identical translations are not listed twice.

LTL formula	# μ/ν variables using LTSmin	# μ/ν variables using BA
$\mathbf{G}(a_y \implies (a_y \mathbf{U} a_r))$	4	2
$\mathbf{G}((a_r \wedge \neg a_y \wedge \neg a_g) \vee (\neg a_r \wedge a_y \wedge \neg a_g)$ $\vee(\neg a_r \wedge \neg a_y \wedge a_g))$	2	1
$\mathbf{G}(a_g \implies \neg \mathbf{X} a_r)$	2	2
$\mathbf{G}((a_g \vee c_g) \implies (b_r \wedge d_r))$	2	1
$\mathbf{GF}a_g$	4	2
$\mathbf{G}(s_a \implies \mathbf{F}\ a_g)$	5	2

Table 1: Results from properties of the traffic light system

The result of the comparison of random formulas can be seen in Figure 11. The list of formulas corresponding to this figure is given in Appendix B. It can be seen that the longer formulas with three or four operators, excluding the **X** operator, result in blow ups in LTSmin. The **X** operator is easier to translate as this only describes the next state. Not all 50 formulas are shown in the graph, as 21 formulas could not be translated by LTSmin. These formulas are too large to be printed, resulting in segmentation fault errors. The longer formulas with 10 variables could not be translated by LTSmin, so no comparison can be shown. These had to be tried one by one, as running them in a list froze my pc when LTSmin resulted in errors. Therefore no comprehensive results are obtained.

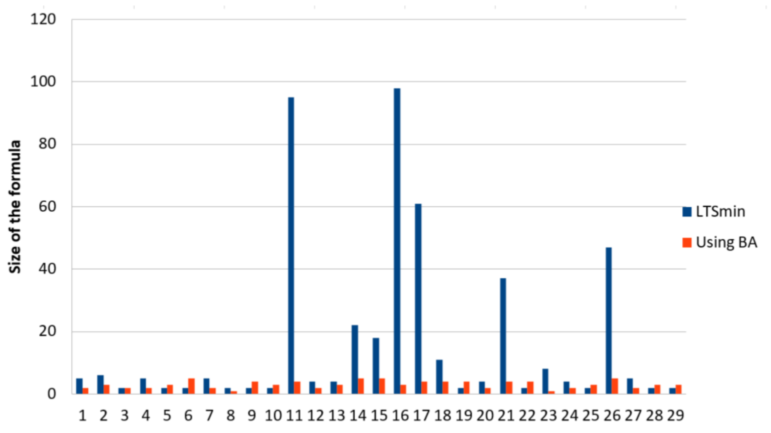

Figure 11: Results from random formulas

Observations The first comparison shows that the new translation produces smaller or equal results for every formula used in the traffic light example. The difference is rather small in some cases as the formulas are also quite short. A more noticeable effect is seen in the comparison of random formulas. About half of the fifty formulas have a huge blow up in size when translated by LTSmin. A few of these can still be printed and those are included in the graph, but most of them are excluded. Despite the fact that they are excluded from the graph, for these cases the translation using Büchi automata brings the most advantage. Finally for the longer formulas the new translation is clearly better, as all of them could be translated using Büchi automata and nearly any could be translated using LTSmin. In a few small formulas the translation of LTSmin is smaller, but this difference is rather small compared to the blow ups in the other cases.

6 Conclusion and future work

The results show that the new translation is significantly better for longer formulas than the translation used in LTSmin. This improvement comes partly from the optimisations performed by Spot and partly from the fact that an equational variant of the μ-calculus is used. Another factor is that LTSmin does not just translate LTL. LTSmin performs a translation from CTL* to

μ-calculus. CTL* is a superset of Computation Tree Logic (CTL) and LTL. CTL is a branching temporal logic and is thus more expressive than LTL.

Originally the idea was to translate TGBA directly to μ-calculus, to avoid the blow up in size by degeneralising TGBA into BA. However, the only method that came to mind was by introducing a μ-calculus variable for each combination of state and acceptance mark. This method is rather similar to the translation using Büchi automata as intermediate form, but lacked the optimisations done by Spot and thus resulted in larger μ-calculus formulas.

For future work improvement may be found by finding a way to translate a TGBA with n states and m acceptance marks into the (equational) μ-calculus, without introducing $n \times (m + 1)$ variables. This could be done by finding a way to describe the infinite behaviour of a set of acceptance marks in single μ-calculus formula, however I am not sure if this is possible.

Another possibility is improving the degeneralisation performed by Spot (Section 3.1). This could be done by analysing different orderings of the levels. Another not yet explored optimisation is the choice of the level of the initial state [8].

References

[1] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008.

[2] Girish Bhat, Rance Cleaveland, and Alex Groce. “Efficient Model Checking Via Büchi Tableau Automata”. In: *Computer Aided Verification, 13th International Conference, CAV, 2001, Paris, France, July 18-22, 2001, Proceedings.* 2001, pp. 38–52.

[3] Julian Bradfield and Colin Stirling. *Modal Logics and mu-Calculi: An Introduction.* 2001.

[4] Julian Bradfield and Igor Walukiewicz. “The mu-calculus and model-checking”. In: *Handbook of Model Checking.* Ed. by H. Veith E. Clarke T. Henzinger. Springer-Verlag, 2015.

[5] Jean-Michel Couvreur. “On-the-Fly Verification of Linear Temporal Logic”. In: *FM’99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I.* 1999, pp. 253–271.

[6] Mads Dam. “CTL* and ECTL* as Fragments of the Modal mu-Calculus”. In: *Theor. Comput. Sci.* 126.1 (1994), pp. 77–96.

[7] Alexandre Duret-Lutz. “Contributions to LTL and ω-Automata for Model Checking”. Habilitation Thesis. Université Pierre et Marie Curie (Paris 6), Feb. 2017.

[8] Alexandre Duret-Lutz. “LTL translation improvements in Spot 1.0”. In: *IJCCBS* 5.1/2 (2014), pp. 31–54.

[9] Alexandre Duret-Lutz et al. “Spot 2.0 - A Framework for LTL and ω-Automata Manipulation”. In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings.* 2016, pp. 122–129.

[10] Gijs Kant et al. “LTSmin: High-Performance Language-Independent Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* 2015, pp. 692–707.

[11] Orna Kupferman and Adin Rosenberg. “The Blowup in Translating LTL to Deterministic Automata”. In: *Model Checking and Artificial Intelligence - 6th International Workshop, MoChArt 2010, Atlanta, GA, USA, July 11, 2010, Revised Selected and Invited Papers.* 2010, pp. 85–94.

Appendices

A Shell scripts

A.1 Count states in Büchi automata

This script reads the number of states in each Büchi automaton, when a list of Büchi automata is provided in the HOA format (output of Spot). It should be used as follows.

```
./countstates.sh filename.hoa
```

The result is put in `filename.count`.

countstates.sh

```
sed -n '/^States:/p' $1 > "${1%.*}".tmp
cut -d' ' -f2- "${1%.*}".tmp > "${1%.*}".count
rm "${1%.*}".tmp
```

A.2 Convert Spot syntax to LTSmin syntax

This script converts the syntax that is used in Spot to the syntax that is used in LTSmin. The input should contain a list of LTL formulas, each on a single line. It should be used as follows.

```
./ltl2ctlstar.sh filename.ltl
```

The result is put in `filename.ctlstar`.

ltl2ctlstar.sh

```
cp $1 "${1%.*}".ctlstar
sed -i 's/1/2/g' "${1%.*}".ctlstar
sed -i 's/0/3/g' "${1%.*}".ctlstar
sed -i 's/\([a-z]\)/\1=="1"/g' "${1%.*}".ctlstar
sed -i 's/!(\([a-z]\)=="1")/\1=="0"/g' "${1%.*}".ctlstar
sed -i 's/G/ [] /g' "${1%.*}".ctlstar
sed -i 's/F/ <> /g' "${1%.*}".ctlstar
sed -i 's/U/ U /g' "${1%.*}".ctlstar
sed -i 's/X/ X /g' "${1%.*}".ctlstar
sed -i 's/&/&&/g' "${1%.*}".ctlstar
sed -i 's/|/||/g' "${1%.*}".ctlstar
sed -i 's/2/true/g' "${1%.*}".ctlstar
sed -i 's/3/false/g' "${1%.*}".ctlstar
sed -i -e 's/^/A (/' "${1%.*}".ctlstar
sed -i -e 's/$/)/' "${1%.*}".ctlstar
```

A.3 Run LTSmin translation for list of LTL formulas

This script runs LTSmin once for each given formula. Afterwards the number of μ and ν variables are counted. A model in ETF format named model.etf should be in the same folder, defining the variables that are being used.

```
./runltsmin.sh filename.ctlstar
```

The result is put in `filename.result`.

runltsmin.sh

```
# Warning: running this program may crash your pc
# if the formulas cannot be translated.
rm -f "${1%.*}".result
rm -f "${1%.*}".success
counter = 0
while read -r line; do
 etf2lts-sym --ctl-star "$line" model.etf -v 2> output.txt
 # cp output.txt formula${counter}.txt
 # Above line can be used to check results
 tail --lines=3 output.txt | head -1 > output2.txt
 cat output2.txt >> "${1%.*}".success
 grep -o 'mu\|nu' output2.txt | wc -l >> "${1%.*}".result
 echo "-------------"
 counter=$((counter+1))
 echo $counter
 echo "-------------"
 cat "${1%.*}".result
done < $1
```

B List of random formulas

Below the formulas are listed that correspond to the result in Figure 11.

1. $\mathbf{F}(\mathbf{G}(\neg(a)))$
2. $\mathbf{F}((b) \vee (\mathbf{G}(\neg(a))))$
3. $\mathbf{F}(b)$
4. $\mathbf{F}(\mathbf{G}(b))$
5. $\mathbf{X}(\mathbf{F}(c))$
6. $\mathbf{X}((a) \vee (\mathbf{X}(\mathbf{X}(\mathbf{F}(\neg(b))))))$
7. $\mathbf{F}(\mathbf{G}(\neg(c)))$
8. $\mathbf{G}(\neg(b))$
9. $\mathbf{X}(\mathbf{X}(\mathbf{X}(\mathbf{G}(\neg(b)))))$
10. $\mathbf{X}(\mathbf{F}(b))$
11. $(\mathbf{F}(\mathbf{G}(\neg(a)))) \vee (\mathbf{G}(a))$
12. $\mathbf{G}(\mathbf{F}(a))$
13. $((\neg(a)) \wedge (\neg(b))) \vee (\mathbf{X}(\mathbf{F}(\neg(b))))$
14. $\mathbf{G}((\mathbf{F}((a) \wedge (b))) \wedge ((a) \vee (\mathbf{F}(\mathbf{G}(c)))))$
15. $\mathbf{X}((c) \vee (\mathbf{F}(\mathbf{G}(b))))$
16. $(\mathbf{F}(b))\mathbf{U}(\mathbf{G}(c))$
17. $\mathbf{F}((c) \vee (\mathbf{G}(\mathbf{F}(\neg(a)))))$
18. $\mathbf{G}((\mathbf{G}(\neg(b)))\mathbf{U}(\mathbf{X}((a) \vee (c))))$
19. $\mathbf{X}(\mathbf{X}(\mathbf{F}(a)))$
20. $\mathbf{G}(\mathbf{F}(\neg(b)))$
21. $(\mathbf{G}(a)) \vee (\mathbf{G}(\mathbf{F}(b)))$
22. $\mathbf{X}(\mathbf{X}(\mathbf{F}(\neg(a))))$
23. $\mathbf{G}((c) \vee ((\mathbf{G}(c))\mathbf{U}(a)))$
24. $\mathbf{G}(\mathbf{F}(\neg(c)))$
25. $(a) \wedge (c) \wedge (\mathbf{F}(\neg(a)))$
26. $((\neg(c)) \wedge (\mathbf{G}(\neg(a))))\mathbf{U}((\neg(a))\mathbf{U}(\neg(b)))$
27. $\mathbf{F}(\mathbf{G}(a))$
28. $\mathbf{X}(\mathbf{F}(\neg(a)))$
29. $\mathbf{X}(\mathbf{X}(\mathbf{G}(\neg(c))))$