LTL Model Checking with use of Generalised Stuttering and Characteristic Patterns

Abdallah Saffidine

September 1, 2009

Abstract

Linear Temporal Logic (LTL) Model Checking can be used to check whether a concurrent system satisfies constraints such as fairness or liveliness among others. The main bottleneck is the space taken by the structure used to represent the system. When the LTL formula does not contain the 'next' operator, partial order reduction can be used to reduce the space requirement. We tried in this internship to use LTL fragments to be able to reduce the space requirement when the formula does contain 'next', or to reduce even more the space requirement when the formula does not contain 'next' and contains a bounded number of 'until'.

First of all, I would like to thank warmly my internship mentor Stefan Schwoon for being always full of advice and ideas. Many thanks to Jan Strejček for sharing his intuition and expertise on the topic. Thanks to Andreas Gaiser, without him the internship would have been much less enjoyable. Finally a warm thanks goes to the whole chair for Foundations of Software Reliability and Theoretical Computer Science, where everybody has been very welcoming.

Foreword

This document is the report of an internship I did in the Chair for Foundations of Software Reliability and Theoretical Computer Science of the Technical University of Munich under Stefan Schwoon's supervision, from February, 23. 2009 to April, 17. 2009. This internship was part of my first year of master studies.

1 Introduction

Model Checking is a fascinating computer science topic at the crossroads of theoretical computer science and hardware or software design. The importance of the subject has been recognised as shown by the 2007 Turing award being received by the inventors of Model Checking. Model Checking, as many verification problems, suffers from the state explosion problem. In Linear Temporal Logic (LTL) Model Checking, the state explosion problem can be partly reduced by using partial order reduction methods. In those methods one tries not to verify every single possible execution of the system but groups of similar executions all at once. This work was an introduction to Model Checking and an attempt to understand partial order reduction in LTL Model Checking. Partial order reduction uses the stutter equivalence relation. The aim of the internship was to try to use the generalised stuttering principle defined in [4] in order to design a generalised partial order reduction method for LTL Model Checking. Although this ambitious goal has not been fully reached, many significant and interesting insights in partial order reduction, generalised stuttering and characteristic patterns were gained.

The report is organised as follows. Section 2 recalls the syntax and semantics of LTL, as well as the definition of the stuttering principle and its use in partial order reduction. Then, in section 3 a first idea to generalise the partial order reduction method is presented with use of generalised stuttering. Finally, in section 4, we define characteristic patterns which can also possibly be used to reduce the space requirement of LTL Model Checking.

2 LTL Model Checking and Partial Order Reduction

LTL Model Checking is a tool to check whether a program, abstracted as a Kripke structure, satisfies properties expressed in a temporal logic called *Linear Temporal Logic* (LTL). The program satisfies the specification when every possible run of the Kripke structure satisfies the formula.

2.1 Linear Temporal Logic

LTL is a modal logic to specify the evolution of a system with respect to time. While a propositional logic formula can be evaluated over sets of variables, an LTL formula is evaluated over sequences of sets of variables. An LTL formula is made of atomic propositions, propositional logic operators ($\neg, \wedge, \vee$) and temporal operators (X *next*, U *until*, G *globally*).

Let σ be a sequence. We define $\sigma(n)$ for $n \in \mathbf{N}$ as the $n+1^{th}$ element of σ. Therefore $\sigma = \sigma(0)\sigma(1)\sigma(2)\dots\sigma(n)\dots$. We also define σ_n to be the n^{th} suffix of σ. The following quality holds: $\sigma_n = \sigma(n)\sigma(n+1)\dots$

Let At be a set of atomic proposition. The LTL formulae on At are defined by induction as $T, p, \neg\phi, \phi \wedge \psi, X\phi, \phi U \psi$ with $p \in At$ and ϕ, ψ two LTL formulae. The semantics of a formula is the set sequences satisfying the formula. For σ a sequence of elements of 2^{At}, we compute by induction the formula satisfied by σ as follows.

- σ satisfies T *(true)*: $\sigma \models T$.
- If p is an atomic proposition, then σ satisfies the LTL formula p if p is set on the first state of σ: $\forall p \in At, \sigma \models p \iff p \in \sigma(0)$.
- If ϕ and ψ are two LTL formulae, then σ satisfies $\phi \wedge \psi$ if σ satisfies ϕ and σ satisfies ψ: $\sigma \models \phi \wedge \psi \iff \sigma \models \phi \wedge \sigma \models \psi$.
- If ϕ is an LTL formula, then σ satisfies $\neg\phi$ if σ does not satisfy ϕ: $\sigma \models \neg\phi \iff \sigma \not\models \phi$.
- If ϕ is an LTL formula, then σ satisfies $X\phi$ (*next* ϕ) if the sequence starting at the second state of σ satisfies ϕ: $\sigma \models X\phi \iff \sigma_1 \models \phi$
- If ϕ and ψ are two LTL formulae, then σ satisfies $\phi U \psi$ (ϕ *until* ψ) if each suffix of σ satisfies ϕ until one satisfies ψ: $\sigma \models \phi U \psi \iff \exists k \in \mathbf{N}, \forall i < k\ \sigma_i \models \phi \wedge \sigma_k \models \psi$

Given a set of LTL formulae A over At, we say that two sequences σ and ρ of $2^A t$ can be *distinguished* in A if there exists a formula satisfied by σ but not satisfied by ρ or if there exists a formula satisfied by ρ but not satisfied by σ.

We defined the basic syntax and semantics of LTL formulae. In practise other operators are useful too. They are defined as abbreviation for the corresponding basic operators combination: $\phi \vee \psi$ abbreviates $\neg(\neg\phi \wedge \neg\psi)$, $F\phi$ reads *finally* ϕ and abbreviates $TU\phi$, $G\phi$ reads *globally* ϕ and abbreviates $\neg F \neg\phi$.

The *nested depth* of X (resp. U) in a formula ϕ and noted $X(\phi)$ (resp. $U(\phi)$) is defined by induction as follows:

- $X(T) = 0$
- $X(\neg\phi) = X(\phi)$
- $X(\phi \wedge \psi) = \max(X(\phi), X(\psi))$
- $X(X\phi) = 1 + X(\phi)$
- $X(\phi U \psi) = \max(X(\phi), X(\psi))$
- $U(T) = 0$
- $U(\neg\phi) = U(\phi)$
- $U(\phi \wedge \psi) = \max(U(\phi), U(\psi))$
- $U(X\phi) = U(\phi)$
- $U(\phi U \psi) = 1 + \max(U(\phi), U(\psi))$

We call $\mathrm{LTL}(U^m, X^n)$ the set of the formulae ϕ such that the nested depth of X (resp. U) in ϕ is less than or equal to n (resp. m). We call $\mathrm{LTL}(U, X^n)$ (resp. $\mathrm{LTL}(U^m, X^n)$) the set of the formulae ϕ with nested depth of X (resp. U) less than or equal to n (resp. m) and any nested depth of U (resp. X).

- $\mathrm{LTL}(U^m, X^n) = \{\phi \in \mathrm{LTL}\ , U(\phi) \leq m \wedge X(\phi) \leq n\}$
- $\mathrm{LTL}(U, X^n) = \{\phi \in \mathrm{LTL}\ , X(\phi) \leq n\}$
- $\mathrm{LTL}(U^m, X) = \{\phi \in \mathrm{LTL}\ , U(\phi) \leq m\}$

2.2 LTL Model Checking

The purpose of Model Checking is to check that a given system behaves accordingly to a specification. The system is represented as a Kripke structure $K = (S, s_i, T, L)$. S is the finite set of states. $s_i \in S$ is the initial state. T is the set of transition relations: $\forall t \in T, t \subset S \times S$. We assume that for each state there is a transition possible from that state: $\forall s \in S, \exists s' \in S, t \in T, (s, s') \in t$ (we write $t(s) = s'$) $L : S \to 2^{At}$ is the labelling function. A path starting at s_0 is a succession of states in S $s_0 s_1 s_2 \ldots s_n \ldots$ respecting the transition relation: $\forall i, (s_i, s_{i+1}) \in T$. We are only considering infinite paths.

The labels of the state represent properties about the system. It can be an excerpt from the memory of the system, and it would thus describe the values of the variables. Otherwise it can be of a higher level nature and reflect easy properties of the system. For instance an atomic proposition could be set when $x > y$ and unset when $x \leq y$. From a given state there may be several transitions enabled because the original system can offer concurrency.

```
cobegin          procedure A()              procedure B()
  A; B;            begin                      begin
coend                for i=1 to 5 do            z = 2;
⋮                      begin                    x = x + 7;
                         x = x + 1;             z = 2 * z;
                         x = x - 1;             z = z - 1;
                       end                    end
                   end
```

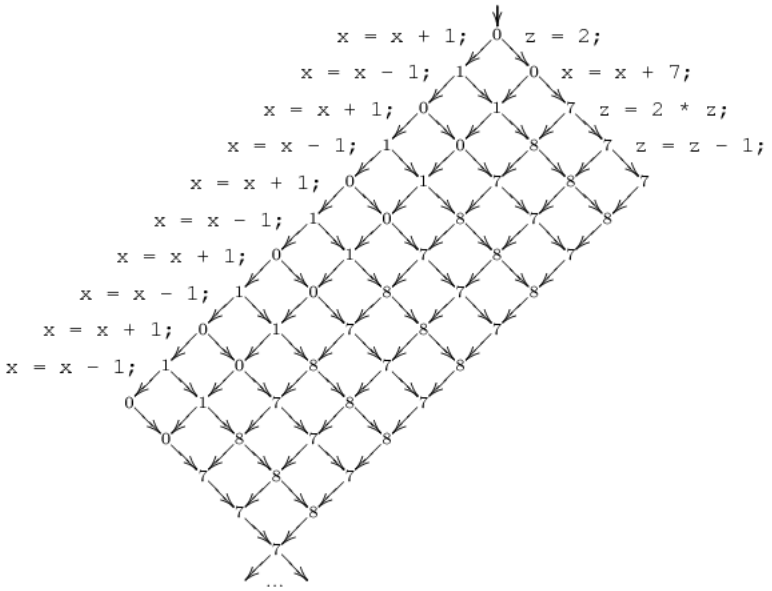

Figure 1: A concurrent program with two threads and the corresponding Kripke structure. We are interested in the value of x at each state of the Kripke structure. Example taken from [4].

For instance fig. 2.2 shows a programme with two parallel threads and the Kripke structure corresponding to the possible evolution from the initial state. Let's have $G(x < 8)$ as specification formula. It involves x only so we keep information about x in each state of the Kripke structure.

Every path $s_0 s_1 \dots s_n \dots$ in K gives rise to a sequence of elements of 2^{At} $L(s_0)L(s_1)\dots L(s_n)\dots$. We say that a LTL formula ϕ is valid for a state s in the Kripke structure K if for every path in K starting at the state s, the labelled sequence corresponding to the path satisfies ϕ.

Initially the system is described implicitly; the verifying system is not provided with the whole Kripke structure but with the starting node corresponding to the initial state of the system and a transition function. The transition function accepts a node as argument and returns the followers

of the node. This procedure allows to verifies the system *on-the-fly*; it is possible to find a counter-example to the specification without unfolding the whole structure.

2.3 Partial Order Reduction

Partial order reduction is a method (or a class of methods) that can be used to decrease the complexity of checking if a system satisfies an LTL formula. Sometimes two sequences are distinct but cannot be distinguished by an LTL formula. Therefore, given a set of formula, equivalence classes for sequences can be drawn. Two sequences are equivalent if they satisfies the same formulae of the set. Partial order reduction is based on this very principle. If two runs of the systems give rises to two equivalent sequences, only one of the two runs needs to be simulated/unfolded.

We are considering the LTL(U, X^0) fragment, that is the 'next'-free formulae. Practical specifications often consist of 'next'-free formulae, mainly due to the fact that the single time step concept is hard to define; it could be a processor step, a step in the algorithm etc.

We start by recalling the stuttering principle which gives us an equivalence relation for sequences such that if two sequences are equivalent then they cannot be distinguished by any LTL(U, X^0) formula.

To define formally the stuttering equivalence we first identify *redundant* letters. Let σ be a sequence. A letter of σ, $\sigma(i)$ is called redundant if $\sigma(i) = \sigma(i+1)$ and there exists $j > i$ such that $\sigma(i) \neq \sigma(j)$. The *canonical form* of σ is the infinite word extracted from σ by removing all redundant words. Two sequences are *stutter equivalent* if they have the same canonical form.

We have the following theorem [4], originally proved by Lamport in 1983.

Theorem 1 *If σ and ρ are stutter-equivalent then they cannot be distinguished in $LTL(U, X^0)$.*

Given a state (or node) in the Kripke structure s, we call $enabled(s)$ the transitions enabled in s. As stated in 2.2, we do not want to unfold the whole Kripke structure. When several runs are equivalent, it is sufficient to check only if one of them satisfies the specification formula. We are going to define $ample(s) \subset enabled(s)$, so that for every run using a transition of $enabled(s)$ there is an equivalent run using a transition of $ample(s)$. Therefore if we prove that all the runs through $ample(s)$ satisfy the specification formula, we will prove that all the runs through $enabled(s)$ satisfy the specification and therefore every run through s satisfies the specification. We define $ample(s)$ conservatively. Of course $ample(s) = enabled(s)$ is possible, but the smaller the ample set, the smaller the explored structure.

We do not give rules to construct explicitly the ample sets, but rather sufficient conditions that the ample sets must fulfil in order to enable all

equivalent classes for the runs. Some heuristic can then suggest sets that are matched against the conditions and taken as the ample set if they follow the conditions.

In the following we assume that we have a Kripke structure $K = (S, s_i, T, L)$, and a specification formula $\phi \in \mathrm{LTL}(U, X^0)$. We assume our labelling function L labels each states only with atomic propositions occurring in ϕ. We want to solve the model checking problem for K and ϕ, that is find a run in K which does not satisfy ϕ or prove that every run in K satisfies ϕ. To be able to compute the ample sets, we use the concepts of *invisibility* and of *independence*. A transition $t \in T$ is said to be *invisible* if it does not change the validity of the atomic proposition: $\forall (s, s') \in t, L(s) = L(s')$. In the example presented fig. 2.2, $z = z * 2$ is an invisible transition as it does not change the value of x. The two transition relations t, t' are *independent* if for every state s in which both are enabled the following holds: $t \in \mathit{enabled}(t'(s)), t' \in \mathit{enabled}(t(s))$ and $t(t'(s)) = t'(t(s))$. In our example the transitions occurring in one thread for procedure A are independent from the transitions occurring in the other for procedure B.

There are four conditions for the ample sets to fulfil [1].

1. $\mathit{ample}(s) = \emptyset$ if and only if $\mathit{enabled}(s) = \emptyset$
2. Along every path in the original structure starting at s it holds that a transition dependent on a transition in $\mathit{ample}(s)$ cannot be executed without a transition in $\mathit{ample}(s)$ occurring first.
3. If $\mathit{ample}(s) \neq \mathit{enabled}(s)$ then every $t \in \mathit{ample}(s)$ is invisible.
4. A cycle is not allowed if it contains a state in which some transition t is enabled, but is never included in $\mathit{ample}(s)$ for any state s on the cycle.

To see if a candidate set fulfils the first condition and the third condition is easy. Invisible transition are stated as such before the algorithm starts, they can be detected by program analysis. When in doubt, it is secure to consider a transition to be always visible. The Kripke structure is explored in a depth-first search manner. This can be used to efficiently match the candidate set against condition four: if the current node is s and one of the transition t would complete a cycle then the node $t(s)$ has already been explored and is still on the stack. We have access to all the nodes of the potential cycle by looking in the stack between $t(s)$ and s, so condition four can be checked. Condition two is actually the hardest to check. In practise we use program/system dependent heuristics to test it.

The partial order reduction has been applied to fig. 2.2 and the result is presented on fig. 2.3. The transitions that did not belong to the ample sets and that were not used to explore the system are drawn with dotted lines.

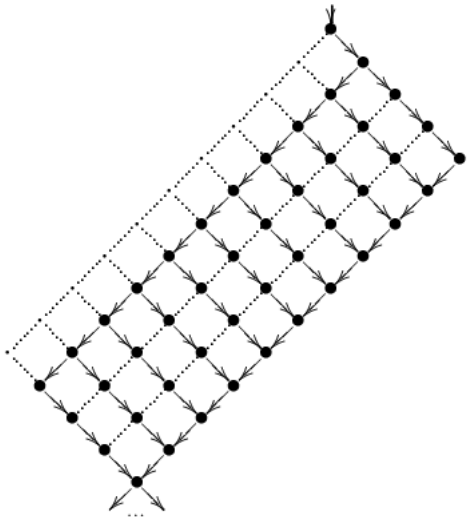

Figure 2: The reduced Kripke structure obtained after using partial order reduction on the example presented in [4] and in fig. 2.2. Dotted lines were not explored. Half of the transitions were not explored and several nodes could be avoided too.

3 Generalised Stuttering

We have seen that when a formula ϕ belongs to LTL(U, X^0), partial order reduction methods through use of the ample sets concept allowed not to unfold the structure of the whole system. The algorithm uses the fact that when two sequences are stutter-equivalent, they cannot be distinguished by ϕ.

Generalised Stuttering was defined in [3] and tries to find better suited[1] equivalence relations when the nested depth of U is small or for formulae that do not belong to LTL(U, X^0).

We call the stuttering equivalence relation previously defined *standard stuttering* in the rest of the report.

3.1 Letter and Subword Stuttering

Standard stuttering states that two sequences are equivalent when they can be reduced to the same sequence of letter by removing consecutive repeated letters. That is, the sequences are equivalent when we do not count the number of adjacent copies for each letter.

Informally n-letter stuttering is a generalisation of standard stuttering

[1] less discriminating, with fewer equivalent classes

that is able to "count" the number of consecutive occurrences for each letter up to $n+1$ but no further. We write $\sigma \simeq_{1,n} \rho$ when σ is n-letter stutter equivalent to ρ. The Standard Stuttering is the 0-letter stuttering.

For instance let $\sigma_0 = aaabbccccaabca^\omega$, $\sigma_1 = aaabbbcccabca^\omega$, $\sigma_2 = aaabbcccaabca^\omega$. We have $\sigma_0 \simeq \sigma_1 \simeq \sigma_2 \simeq abcabca^\omega$, but $\sigma_0 \simeq_{1,1} \sigma_2 \not\simeq_{1,1} \sigma_1$, and $\sigma_0 \not\simeq_{1,3} \sigma_2$.

To define formally the n-letter stuttering, we extend concepts of the standard stuttering. For a given sequence σ, a letter $\sigma(i)$ is n-redundant if it occurs at least $n+1$ times consecutively $\sigma(i) = \sigma(i+1) = ... = \sigma(i+n+1)$ and but no infinitely many times consecutively $\exists j > i, \sigma(i) \neq \sigma(j)$. The n-canonical form of σ is extracted from σ by removing every n-redundant letter. Two sequences are n-letter stutter equivalent if they have the same n-canonical form.

Remark that if $\sigma \simeq_{1,n} \rho$ then $\forall 0 \leq i \leq n, \sigma \simeq_{1,i} \rho$. The $n+1$-letter stuttering equivalence is refining the n-letter stuttering. Strejček proved that LTL(U, X^n) formulae could not distinguish between n-letter stutter equivalent sequences [3], generalising theorem 1.

The m-subword stuttering is another generalisation of the standard stuttering. Here we do not only delete consecutively repeated letters to obtain a canonical form but also whole words. For instance $\sigma_0 = ababab c^\omega$ and $\sigma_1 = abab c^\omega$ are 0- and 1-subword stutter equivalent but not 2-subword stutter equivalent. The repeated subword is ab.

Strejček proved in [3] that LTL(U^m, X^0) formulae could not distinguish between m-subword stutter equivalent sequences. A formal definition of the m-subword stuttering, as well as a broader generalisation of the stuttering principle are presented in [3].

3.2 Generalised Partial Order Reduction

Fig. 3.2 shows a hand-written partial order reduction for the example presented in fig. 2.2. No algorithm is known yet to obtain such a result automatically.

We present an argument (but no formal proof) to show that partial order reduction based on generalised stuttering can be computationally expensive. We develop an example based on the 1-letter stuttering but it is easy to extend to n-letter stuttering and m-subword stuttering. This extension is straightforward because of the counting aspect that n-letter stuttering and m-subword stuttering share and which does not occur in the standard stuttering.

The examples provided in fig. 3.2 and 3.2 show two Kripke structure and a minimal partial order reduced structure for 1-letter stuttering. Let call a the label with horizontal strips, b the label with black filling, and c the label with vertical strips. In fig. 3.2 the possible sequences of labels are $baaabbbbbba$, $baaaabbbbba$, ... and $baaaaaaabba$. They all have the

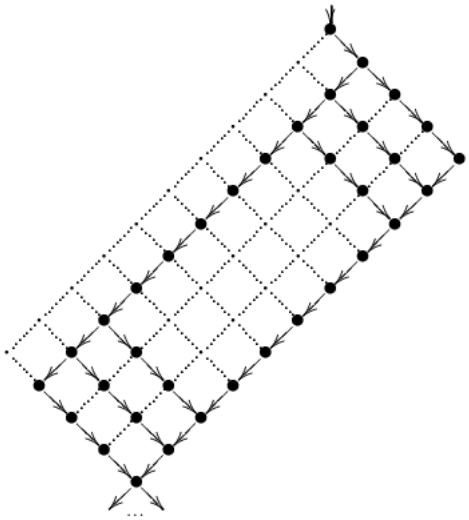

Figure 3: Running example (fig. 2.2, 2.3) reduced with use of the general stuttering principle. The reduction is done by hand.

same 1-canonical form *baabba*. Therefore only one run is needed to check whether a LTL(U, X^1) formula is satisfied. In the reduced structure, one run is possible and it has *baabba* as 1-canonical form so both structure are equivalent with respect to LTL(U, X^1) formulae satisfaction. In the system presented in fig. 3.2 however, there are several possible canonical sequences: *baabbcca*, *baabbca*, and *baabca*. The reduced structure allows runs for every canonical sequence. The problem is that it is impossible to know locally if a transition should be omitted or not. In our example, one has to look ahead to enable a transition to the right soon enough so that the canonical sequence *baabbcca* remains reachable. The look-ahead distance is linear in n in case of n-letter stuttering and in $m \times \mu$ in case of m-subword stuttering with a word of size μ being repeated. The main bottleneck is that the look-ahead distance is also linear in the number of transitions independent from the invisible one. Here we have two transitions going to the left and we are dealing with 1-letter stuttering so the look-ahead distance is about 2×1.

The second example 3.2 show that it is possible to construct structure that need to look ahead arbitrarily far ahead. There are also structures that need to be able to look far backward among the predecessors of the node, not only on the stack while searching but among all predecessors within a given arbitrarily profound depth. Those backward structures were not introduce for the sake of brevity.

4 Characteristic Patterns

We have seen that the generalised stuttering equivalence relation was connected to the LTL hierarchies in the following way. If σ and ρ are m,n-stutter equivalent, then they cannot be distinguished by any LTL(U^m, X^n) formula. However for all m, n there exist sequence σ and ρ that are not m,n-stutter equivalent but cannot be distinguished by any LTL(U^m, X^n) formula [4]. It means that in general the best reduction achievable through generalised stuttering is not as good as the optimal reduction.

The *characteristic patterns* were defined to match the LTL hierarchy better. For all m, n a set of characteristic (m, n)-patterns is defined. Each sequence is represented by exactly one pattern. Two sequences are represented by the same characteristic (m, n)-pattern if and only if they cannot be distinguished by any LTL(U^m, X^n) formula.

For the sake of brevity and simplicity, we will not present here the general (m, n)-patterns but only $(m, 0)$-patterns. For a more complete and formal introduction to characteristic patterns, the seminal paper by Strejček [2] is advised.

4.1 Intuitive Definition

To get an idea of how characteristic patterns are constructed, let's consider the following construction of the $(2, 0)$-pattern of a given word. We write $(1, 0)$-patterns made out of an alphabet Σ with a succession of letters of Σ in parenthesis. Let's have $\Sigma = \{a, b, c\}$ as an alphabet. Let $\alpha = aabaca^\omega \in \Sigma^\omega$ be a ω-word over the alphabet Σ. The $(1, 0)$-pattern of α, denoted $pat(1, 0, \alpha)$ is the finite word obtained from α by deletion of all repeated letter. $pat(1, 0, \alpha) = (abc)$. Recall that α_n is the n^{th} suffix of α. We can compute the $(1, 0)$-patterns of the suffixes of α. For instance $pat(1, 0, \alpha_1) = (abc)$, $pat(1, 0, \alpha_2) = (bac)$, $pat(1, 0, \alpha_3) = (ac)$. The sequence of $(1, 0)$-patterns of α is $(abc)(abc)(bac)(ac)(ca)(a)^\omega$, it is called $patword(1, 0, \alpha)$. $patword(1, 0, \alpha)$ is an ω-word over the alphabet of the possible $(1, 0)$-patterns of Σ : $Pats(1, 0, \Sigma)$.

The $(2, 0)$-pattern of α is obtained by removing the repeated patterns of $patword(1, 0, \alpha)$. If we look at $patword(1, 0, \alpha)$ as an ω-word of $Pats(1, 0, \Sigma)$ we have that $pat(2, 0, \alpha)$ the $(2, 0)$-pattern of α is a $(1, 0)$-pattern in the alphabet $Pats(1, 0, \Sigma)$. $pat(2, 0, \alpha) = ((abc)(bac)(ac)(ca)(a)) \in Pats(2, 0, \Sigma)$

Formally for $\alpha \in \Sigma^\omega$ we can define by induction over $m \in \mathbf{N}$, $pat(m, 0, \alpha)$ the characteristic $(m, 0)$-pattern of α and $patword(m, 0, \alpha)$ the $(m, 0)$-pattern word of α as follows [2]:

- $pat(0, 0, \alpha) = \alpha(0)$,
- $patword(m, 0, \alpha) \in Pats(m, 0, \Sigma)^\omega$ such that $patword(m, 0, \alpha)(i) = pat(m, 0, \alpha_i)$,

- $pat(m+1, 0, \alpha)$ is the finite word obtained form $patword(m, 0, \alpha)$ by deletion of all repeated letters.

4.2 Usage of Characteristic Patterns for Model Checking

We define (m, n)-pattern, noted $\sim_{m,n}$, equivalence relation as follows: $\sigma \sim_{m,n} \rho$ if and only if σ and ρ have the same (m, n)-pattern. Characteristic patterns are linked to LTL formulae through the following result:

Theorem 2 *For all sequences σ, ρ, σ and rho cannot be distinguished in LTL(U^m, X^n) if and only if they are equivalent, $\sigma \sim_{m,n} \rho$.*

Therefore we can say that a pattern $p \in Pats(m, n, \Sigma)$ satisfies a formula $\phi \in$LTL(U^m, X^n), written $p \models \phi$, if for every sequence σ such that $pat(m, n, \sigma) = p$, $\sigma \models \phi$. Moreover it is possible to directly check whether a pattern p satisfies a formula ϕ by using the following procedure.

Algorithm 1 An algorithm checking whether $p \models \phi$ by induction on p and ϕ. If $p \in Pats(m, n, \Sigma)$ then $mtype(p) = m$.

```
check(φ, p, n : int) : bool
  if U(φ) < mtype(p) then
    return check(φ, p(0), n)
  else if φ = T then
    return true
  else if φ ∈ Σ then
    return φ = p(n)
  else if φ = ¬ψ then
    return ¬ check(ψ, p, n)
  else if φ = ψ1 ∧ ψ2 then
    return check(ψ1, p, n) ∧ check(ψ2, p, n)
  else if φ = Xψ then
    return check(ψ, p, n + 1)
  else if φ = ψ1 U ψ2 then
    i ← 0
    while i < |p| ∧ ¬ check(ψ2, p, n) do
      if check(ψ1, p(i), n) then
        i ← i + 1
      else
        i ← |p|
    return i < |p|
```

We tried to use the characteristic patterns for Model Checking. The principle was to compute the patterns that can occur in a given Kripke structure. The patterns would be generated on the fly and it would be possible to see whether all of them satisfied the formula. When one pattern

does not satisfy the formula, we know that the specification is not fulfilled by the structure. Even if it is not possible to generate the patterns on the fly, it could still be an improvement over standard LTL Model Checking in some cases. Standard LTL Model Checking and partial order reduction take indeed [2] as input the product of the Kripke structure with the Büchi automaton representing the negation of the LTL formula to be checked. In general the Kripke structure is already huge and the automation does not add much to the space complexity, but in some pathological cases, being able to directly match the specification against the structure without computing the cross product with Büchi automaton is valuable.

It has been possible to compute the $(1, 0)$- and $(2, 0)$-patterns of the Kripke structure, but no general algorithm to find efficiently the (m, n)-patterns with $m > 2$ has been fully designed yet.

5 Conclusion

LTL Model Checking is sometimes tractable thanks to the partial order reduction with ample sets method. This partial order reduction method is based on the LTL(U, X^0) fragment of Linear Temporal Logic. Therefore this method cannot deal with specification formulae using the 'next' operator, and when only a few 'until' operators are used, one could hope for a better reduction. In this internship we have started to explore two ideas from [4] to design a general partial order reduction method for the fragments of the LTL(U^m, X^0), LTL(U, X^n) and possibly LTL(U^m, X^n) hierarchies. One possible way was to use the concept of generalised stuttering and try to find a generalisation of the ample sets rules, but it was hinted that knowing when to omit a transition could be intractable due to look-ahead requirements. A different way has also been envisioned, it was shown that computing the characteristic patterns of a Kripke structure would make it possible to alleviate the state explosion problem in certain cases, but no general algorithm to compute the characteristic patterns of any Kripke structure has been found yet.

[2] This schemata was omitted in the report for the sake of brevity and simplicity.

References

[1] E. M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. 1999.

[2] Antonín Kucera and Jan Strejcek. Characteristic patterns for ltl. In Peter Vojtás, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, *SOFSEM*, volume 3381 of *Lecture Notes in Computer Science*, pages 239–249. Springer, 2005.

[3] Antonín Kucera and Jan Strejcek. The stuttering principle revisited. *Acta Inf.*, 41(7-8):415–434, 2005.

[4] Jan Strejček. *Linear Temporal Logic: Expressiveness and Model Checking*. PhD thesis, Faculty of Informatics, Masaryk University, Brno, Czech Republic, 2004.

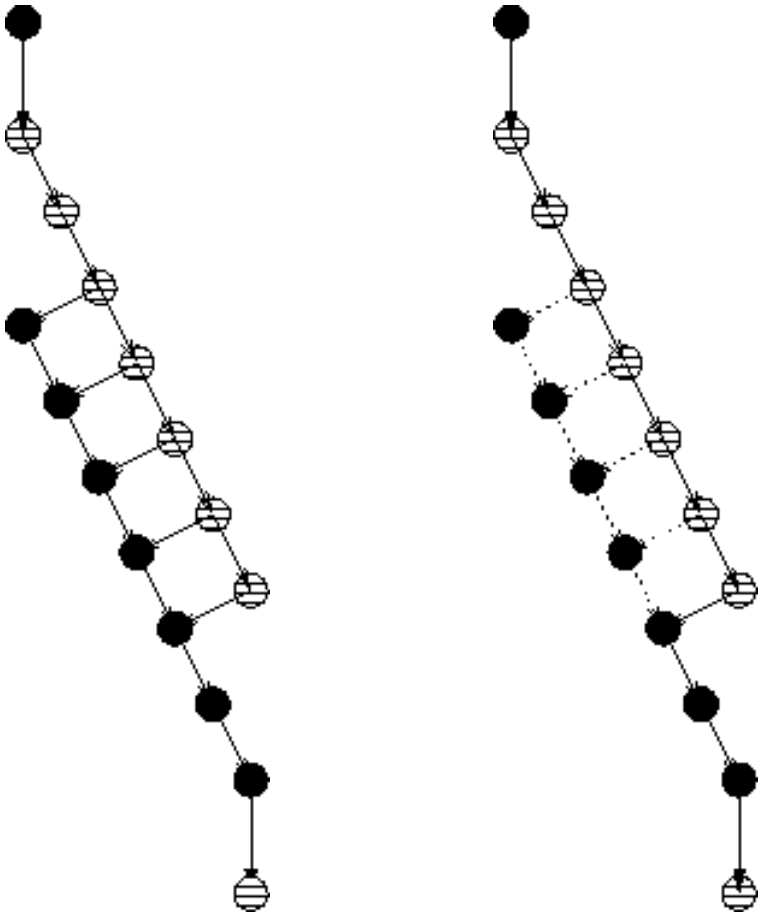

Figure 4: Example of partial order reduction for 1-letter stuttering. *Left* original Kripke structure, *right* minimal reduced Kripke structure. Transitions going from left to right are invisible and transition going from right to left are possibly visible. Transitions going in different directions are independent.

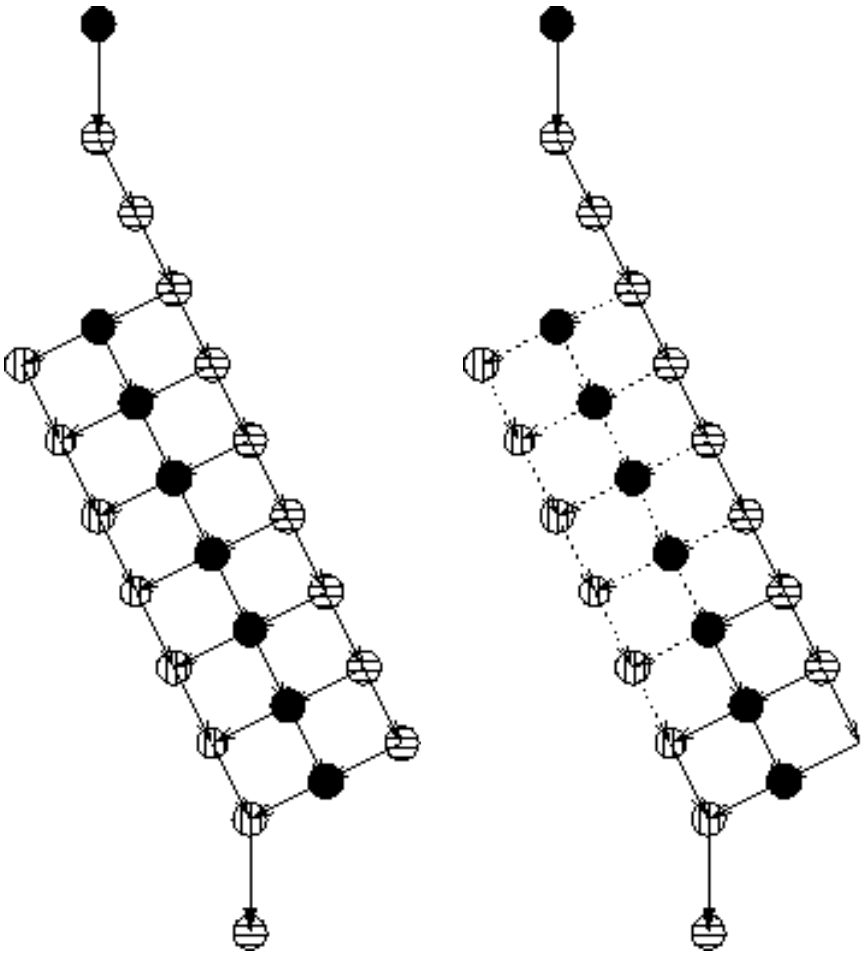

Figure 5: Example of partial order reduction for 1-letter stuttering. *Left* original Kripke structure, *right* minimal reduced Kripke structure. We can see on the reduce Kripke structure that global factors determine which transitions are needed in to fully represent the original system. Transitions going from left to right are invisible and transition going from right to left are possibly visible. Transitions going in different directions are independent.