On model-checking for fragments of μ-calculus

E. A. Emerson[1⋆] and C. S. Jutla[2] and A. P. Sistla[3 ⋆⋆]

[1] Department of Computer Science, University of Texas at Austin, Austin, Texas
[2] I.B.M. Thomas J. Watson Research Laboratories
[3] Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL 60680.

1 Introduction

In this paper we consider the problem of model checking for different fragments of propositional μ-calculus. This logic was studied by many authors [4, 8] for specifying the properties of concurrent programs. It has been shown (see [12, 10, 5]) to be as expressive of automata on infinite trees. Most of the known temporal and dynamic logics can be translated into this logic.

The model checking problem for this logic was first considered in [6]. In that paper, the authors presented an algorithm that is of complexity $O((mn)^{l+1})$ where m is the length of the formula, n is the size of the Kripke structure and l is the number of alternations of least and greatest fixed points in the given formula. Thus the complexity of the algorithm is exponential in the length of the formula. Since then there have been other algorithms [1, 3, 11] that were presented. Although some of these algorithms have lower complexity than the original algorithm, their complexity is still exponential. Algorithms of linear complexity (both in the size of the structure and the formula) were given [2] for the case when there is no alternation of least and greatest fixed points in the given formula.

In this paper, we consider the model checking problem for different fragments of the μ-calculus. We first consider two fragments called L_1, L_2 and give model checking algorithms for these fragments which are of complexity $O(m^2n)$ where m is the length of the formula and n is the size of the structure. The formulas in L_1 and L_2 allow arbitrary nesting of the least and greatest fixed points. However, they restrict how the modal operators and the boolean connectives can appear in the formula. The fragment L_2 is shown to be exactly as expressive as the branching time temporal logic ECTL* considered in [13]. ECTL* is the extended version of CTL* in which the path formulas have the same expressive power as ω-regular expressions.

We also consider the model checking problem for the full μ-calculus and show that this problem is equivalent to the non-emptiness problem of parity tree automata considered in [9, 5]. More specifically, we show that the model checking problem for μ-calculus is reducible to the non-emptiness problem for

[1] This author's research is supported in part by the ONR grant N00014-89-J-1472 and Texas Advanced Technology Program Grant 003658-250.
[2] This author's research is supported in part by NSF grant CCR-9212183.

parity tree automata of size $O(mn)$ where m and n are as defined above. We also show that the non-emptiness problem of parity tree automata of size p is reducible to the model checking problem for μ-calculus in which the size of the Kripke structure is $O(p)$ and the length of the formula is $O(p)$. This shows that there is an efficient algorithm for one of them iff there is such an algorithm for the other.

The paper is organized as follows. Section 2 contains definitions and notation. Section 3 presents the model checking algorithms for the logics L_1 and L_2. Section 4 contains the result showing the equivalence of the model checking problem for the full μ-calculus and non-emptiness problem for the parity tree automata.

2 Definitions and Notation

In this section we define the syntax and semantics of the different fragments of the logic μ-calculus. Let $\mathcal{P}$ and $\mathcal{X}$ be two disjoint sets of elements. The elements of $\mathcal{P}$ will be called *atomic propositions* and are usually denoted by $P, Q, ...$ The elements of $\mathcal{X}$ will be called variables and are usually denoted by $x, y, ...$ The formulae of μ-calculus are formed using the symbols from $\mathcal{P}$, $\mathcal{X}$, the propositional connectives $\neg$ and $\wedge$, the modal operator `<R>`, and the symbol μ.

The set of well-formed formulas of μ-calculus are defined inductively. The symbols `true` and `false` are well-formed formulas. Every atomic proposition and every variable are well-formed formulas. If f and g are well-formed formulas then $\neg f$, $f \wedge g$ and `<R>`f are also well-formed formulas. In addition, if f is well-formed formula in which all the occurrences of the variable x are in the scope of an even number of negations then $\mu x(f)$ is also a well-formed formula.

We say that a variable x is free variable in a formula f if there is an occurrence of x in f which is not in the scope of some μx. Let *free-var*(f) denote all the variables that are free in f. A variable which appears in f and which is not free, is called a bound variable. A formula without any free variables is called a *closed* formula. A formula that has no variables will be called a constant. We define the semantics of the formulas in μ-calculus with respect to a *kripke structure*. A kripke structure K is a triple (S, R, L) where S is a finite set of states, $R \subseteq S \times S$ is a total binary relation (i.e. $\forall x \exists y (x, y) \in R)$), and $L : S \rightarrow 2^{\mathcal{P}}$. With each state s, L associates a set of atomic propositions that are true in that state. Let f be a formula with free-var$(f) = \{x_1, ..., x_k\}$. An evaluation ρ for f is a mapping that associates with each variable in free-var(f) a subset of S. If free-var(f) is empty then there is a unique empty evaluation ϵ for f. For a given kripke structure K, we define a function $\mathcal{M}_{(K,f)}$ from the set of evaluations for f to the subsets of S, by induction on the structure of f as follows.

- $\mathcal{M}_{(K,P)}(\epsilon) = \{s : P \in L(s)\}$ where P is an atomic proposition;
- $\mathcal{M}_{(K,f \wedge g)}(\rho) = \mathcal{M}_{(K,f)}(\rho') \cap \mathcal{M}_{(K,g)}(\rho'')$ where ρ' and ρ'' are restrictions of ρ to the free variables of f and g respectively;
- $\mathcal{M}_{(K,\neg f)}(\rho) = S - \mathcal{M}_{(K,f)}(\rho)$;

- $\mathcal{M}_{(K,\texttt{<R>}f)}(\rho) = \{s : \exists s' \in \mathcal{M}_{(K,f)}(\rho) \text{ such that } (s,s') \in R\}$;
- $\mathcal{M}_{(K,\mu x f)}(\rho) = \bigcap\{S' \subseteq S : S' = \mathcal{M}_{(K,f)}(\rho')$ where $\rho'(x) = S'$ and for all other $y \in$ free-var(f), $\rho'(y) = \rho(y)\}$.

In the above definition, it is to be noted that the value of $\mathcal{M}_{(K,\mu x f)}(\rho)$ is given as a least fixed point. For a closed formula f, we say that a state s in K satisfies f (written as $K, s \models f$) iff $s \in \mathcal{M}_{(K,f)}(\epsilon)$. We define derived connectives defined as follows: $f \vee g \equiv \neg(\neg f \wedge \neg g)$, $f \rightarrow g \equiv (\neg f \vee g)$, $\texttt{[R]}f \equiv \neg\texttt{<R>}\neg f$, $\nu y f(y) \equiv \neg\mu x(\neg f(\neg x))$. It is to be noted that while μx denotes the least fixed point νy denotes the greatest fixed pont operator.

By using DeMorgan's laws, the identities $\neg\nu y f(y) \equiv \mu x(\neg f(\neg x))$ and $\neg\texttt{[R]}f \equiv \texttt{<R>}\neg f$, we can transform any formula into an equivalent formula in which all negations apply only to the atomic propositions. Such formulas will be called normalized formulas. In our paper we will be interested in these types of formulas. A formula of the form $\mu x f$ (resp., $\nu x f$) will be called a μ-formula (resp., ν-formula).

With a normalized formula f, we define an integer $altdepth(f)$. Intutively, $altdepth(f)$ will be the maximum number of alternations of μs and νs in f. Formally, it is defined as follows.

- For a μ-formula f, $altdepth(f) = 0$ if f has no ν-subformulas in it, otherwise $altdepth(f) = 1 + max\{altdepth(g) : g \textit{ is a } \nu\textit{-subformula of } f\}$.
- For a ν-formula f, $altdepth(f) = 0$ if f has no μ-subformulas in it, otherwise $altdepth(f) = 1 + max\{altdepth(g) : g \textit{ is a } \mu\textit{-subformula of } f\}$.
- For any formula f, define $altdepth(f) = 1 + max\{altdepth(g) : g$ *is* $a\mu$-subformula or a ν-subformula of $f\}$.

We assume throughout the paper that each variable appearing in a formula is bound at most once. This means that we can not have two subformulas of the form $\mu x(g)$ and $\mu x(h)$ appearing in a formula. If this property is not satisfied, then by renaming the variables we can obtain an equivalent formula that satisfies this property.

For finite kripke structures, the least fixed point can be computed by iteration starting with an empty set and iterating until a fixed point is reached. Similarly, the greatest fixed point can be computed by starting from the set containing all states and iterating until a fixed pont is reached. These results are due to Tarski/Knaster.

Now, we define two fragments of the μ-calculus L_1 and L_2 defined as follows. The set of L_1 formulas are exactly those that are formed using the following rules:

1. All the members of $\mathcal{P} \cup \mathcal{X}$ are L_1-formulas; i.e. all atomic propositions and all variables are L_1-formulas.
2. If f is a L_1-formula that does not have any variables appearing in it then $\neg f$ is also an L_1-formula.
3. If f and g are L_1-formulas then $f \vee g$, $\texttt{<R>}f$, $\mu x(f)$ and $\nu x(f)$ are L_1-formulas.
4. If f and g are L_1-formulas such that at most one of them has variables appearing in it, then $f \wedge g$ is a L_1-formula.

Rule 2 states that negations can only be applied to formulas which are constants. Rule 4 states that if we have a conjunction only one of the conjuncts can have variables and all other conjuncts have to be constants. Due to these restrictions, when we normalize an L_1-formula, i.e. push the negations down, the resulting formula has the following property: all the `[R]`s only apply to constant formulas, and in any conjunction only one of the conjuncts can have variables appearing in it. This means that an L_1-formula is almost like a linear-time formula.

Let L_2 be the set of formulas obtained by using rules 2a and 4a, given below, in place of 2 and 4.

2a. If f is a closed L_2-formula then $\neg f$ is also an L_2-formula.
4a. If f and g are L_2-formulas such that at most one of them is an open formula then $f \wedge g$ is a L_2-formula.

It is to be noted that the formula f in rule 2 should be a constant formula while in rule 2a it can be any closed L_2-formula. Similarly, in rule 4, at least one of f and g has to be a constant, while in 4a, at least one of them has to be a closed formula. As a consequence, rules 2 and 4 are special cases of rules 2a and 4a respectively. From this, it should be easy to see that L_1 is a subset of L_2. The expressive power of L_2 is characterized by theorem 3.2 given in the next section.

3 Model Checking for the restricted Logics

In this section, we present efficient procedure for model-checking for the two logics L_1 and L_2. First, we consider the logic L_1 and present an efficient model-checking algorithm for this logic. This algorithm, as we show later, can be easily extended to the logic L_2.

Let f be a closed normalized L_1-formula. Let $SF(f)$ denote the set of subformulas of f. The set $SF(f)$ can be defined inductively. Let $K = (S, R, L)$ be a given kripke structure. We define a graph $G_{K,f} = (V, E)$, where V is the set of vertices and E is the set of edges, defined as follows. The node set $V = \{(s, g) : s \in S, g \in SF(f)\}$. Essentially, there is one node in V corresponding to each state in S and each subformula of f. The set of edges leaving the node (s, g) are, defined according to the outermost connective of the subformula g, as follows.

- If $g = P$ or $g = \neg P$ where P is an atomic proposition then there are no edges leaving (s, g).
- If $g = x$ where x is variable and g' is the largest subformula of f such that $g' = \mu x(g'')$ or $g' = \nu x(g'')$, then there is exactly one edge leaving (s, g) and this edge is to (s, g').
- If $g = \mu x(g')$ or $g = \nu x(g')$, then there is an edge from (s, g) to (s, g') and this is the only edge from (s, g).
- If $g = g' \wedge g''$ or $g = g' \vee g''$, then there are two edges from (s, g), to the nodes (s, g') and (s, g'').

– If $g = \texttt{<R>}g'$ or $g = \texttt{[R]}g'$, then for each state s' such that $(s, s') \in R$, there is an edge from (s, g) to (s', g').

A path in $G_{K,f}$ is a finite sequence of nodes such that there is an edge in E from each node in the path to the succeeding node. A path starting and ending with the same node is a cycle. A subformula $g \in SF(f)$ is called a μ-subformula (respectively, a ν-subformula) if g is of the form $\mu x(f)$ (respectively, $\nu x(f)$). We say that a cycle C in $G_{K,f}$, is a ν-cycle (respectively, μ-cycle) if the longest subformula appearing in a node on C is ν-subformula (respectively, μ-subformula). A node (s, g) in $G_{K,f}$ is called a $\wedge$-node if g is of the form $g_1 \wedge g_2$ or is of the form $\texttt{[R]}g_1$; note that in the later case g_1 will be a constant. A node (s, g) is called a $\vee$-node if g is of the form $g_1 \vee g_2$ or is of the form $\texttt{<R>}g_1$.

Observation 3.1 *The graph $G_{K,f}$ satisfies the following properties.*

- *Assume that there is an edge from (s, g) to (s', g') in $G_{K,f}$.*
  - *If $g = \texttt{<R>}g'$ or $g = \texttt{[R]}g'$ then $(s, s') \in R$; otherwise, $s' = s$.*
  - *If g is not a variable then g' is a subformula of g. If g is a variable then g is a subformula of g'.*
- *For any node (s, g) in $G_{K,f}$, there is a path from (s, g) to a node on a cycle iff g has at least one variable in it (i.e. g is not a constant).*
- *Let C be a cycle and (s, g) be a node on it such that g is the longest formula appearing in all the nodes on C. Then, g is a μ-subformula or a ν-subformula. In addition, all other subformulas appearing in some node on C themselves are subformulas of g.*

Now, we label the nodes of $G_{K,f}$ as follows. With each node $u \in V$, we maintain a variable $label(u)$ that denotes the label of the node u. Each of these variables takes one of the three values—true, false, NIL, and is initialized to the value NIL. During the execution of the algorithm, the values of these variables will be set to true or false. When once a variable is set to one of these two values, it will never be changed. Furthermore, for any node $u = (s, g)$, at the end of the execution of the algorithm, $label(s, g) = \mathsf{true}$ iff $K, s \models g$.

At any time during the execution of the algorithm, if $label(u) = NIL$ then we say that node u is *unlabeled* at that time. We say that a path is unlabeled if all the nodes on the path are unlabeled. Let n be the length of the formula f. We execute the following algorithm on the graph $G_{K,f}$.

1. For each node $u \in V$, $label(u) \leftarrow NIL$.
2. For each $g \in SF(f)$ in increasing lengths of g, and for each $s \in S$, update $label(u)$, where $u = (s, g)$, as follows.
   - $g = P$: *If* $P \in L(s)$ *then* $label(u) \leftarrow \mathsf{true}$ *else* $label(u) \leftarrow \mathsf{false}$.
   - $g = \neg P$: *If* $P \notin L(s)$ *then* $label(u) \leftarrow \mathsf{true}$ *else* $label(u) \leftarrow \mathsf{false}$.
   - $g = g' \wedge g''$ or $g = \texttt{[R]}g'$:
     If for all successors u' of u it is the case that $label(u') = \mathsf{true}$ *then* $label(u) \leftarrow \mathsf{true}$;
     If for some successor u' of u such that $label(u') = \mathsf{false}$ *then* $label(u) \leftarrow \mathsf{false}$. In other cases, $label(u)$ is unchanged (i.e. $= NIL$).

- $g = g' \vee g''$ or $g = $ <R>g' :
  If for some successor u' of u $label(u') = \mathsf{true}$ *then* $label(u) \leftarrow \mathsf{true}$;
  If for all successors u' of u it is the case that $label(u') = \mathsf{false}$ *then* $label(u) \leftarrow \mathsf{false}$.
  In other cases, $label(u)$ is unchanged.
- None of the above: $label(u)$ is unchanged.

3. *For* each unlabeled node $u \in V$, *if* there exists an unlabeled path from u to an unlabeled ν-cycle *then* $label(u) \leftarrow \mathsf{true}$.
4. *For* each unlabeled node u, $label(u) \leftarrow \mathsf{false}$.

Theorem 3.1 *After the execution of the above algorithm, for any node* $u = (s, g)$ *in* $G_{K,f}$ *where* g *is a closed subformula,* $label(u) = \mathsf{true}$ *iff* $K, s \models g$.

In order to prove the above theorem, we need some lemmas. First, it is to be noted that after the execution of step 2 of the above algorithm, the following conditions are satisfied. For all nodes u in $G_{K,f}$ such that there is no path connecting u to a node on a cycle, $label(u) \neq NIL$. Hence, for each node $u = (s, g)$ where g is a constant, $label(u) \neq NIL$. For this case, it should be easy to see that $label(s, g) = \mathsf{true}$ iff $K, s \models g$. Also, for every node $u = (s, g)$ such that $label(s, g) = NIL$, there is at least one successor node u' such that $label(u') = NIL$. In addition, if $g = g' \wedge g''$, then for one successor u', $label(u') = \mathsf{true}$ and for the other successor u'', $label(u'') = NIL$. Consider any node (s, g) such that $label(s, g) \neq NIL$. For any evaluation ρ over the free variables of g, the following property holds: $s \in \mathcal{M}_{K,g}(\rho)$ iff $label(s) = \mathsf{true}$. For any node (s, g) where g is a μ-formula or a ν-formula, $label(s, g) = NIL$.

In step 3 of the algorithm, all nodes lying on unlabeled ν-cycles or all nodes that have unlabeled paths to such cycles will be labeled true. Now, we prove that step 3 of the above algorithm is sound. To do this, we need the following lemmas.

Lemma 1. *Let* $u_0, u_1, ..., u_k$ *be a path in* $G_{K,f}$ *after execution of step 2 of the above algorithm where for each* $i = 0, 1, ..., k-1$, u_i *is a* $\wedge$*-node or a* $\vee$*-node. Also, let* $u_i = (s_i, g_i)$. *Then, each* g_i *is a subformula of* g_0*, and for any evaluation* ρ *on the free-var*(g_0)*, if* $s_k \in \mathcal{M}_{K,g_k}(\rho')$ *where* ρ' *is a restriction of* ρ *to free-var*(g_k)*, then* $s_0 \in \mathcal{M}_{K,g_0}(\rho)$.

The above lemma can be proved by a simple induction on the length of g_0.

Lemma 2. *Let* $u_0, u_1, ..., u_k$ *be an unlabeled path in* $G_{K,f}$ *after the execution of step 2 of the above algorithm satisfying the following conditions: for each* $i > 0$, $u_i = (s_i, g_i)$, g_i *is a strict subformula of* g_0 *and* $g_k \in$*free-var*(g_0). *Then, for any evaluation* ρ *over free-var*(g_0)*, if* $s_k \in \rho(g_k)$ *then* $s_0 \in \mathcal{M}_{K,g_0}(\rho)$.

The following lemma shows that step 3 of the above algorithm is sound.

Lemma 3. *Let*

- $u_0, u_1, ..., u_k = u_0$ *be an unlabeled ν-cycle in $G_{K,f}$ after execution of step 2 of the above algorithm;*
- $u_i = (s_i, g_i)$ *for $0 \leq i < k$, where $g_0 = \nu x(g')$ is the longest subformula appearing in any node of the cycle;*
- $S' = \{s_i : 0 \leq i < k, g_i = g_0\}$.

Then, for any evaluation ρ on the free-var(g_0), $S' \subseteq \mathcal{M}_{K,g_0}(\rho)$.

The remainder of the proof of theorem 3.1 and the proofs of the lemmas will be given in an extended version of this paper.

Complexity and Expressiveness

Below, we discuss the complexity of the above algorithm. First, it is to be noted that the number of vertices in $G_{K,f}$, i.e. $|V|$, is $O(|S||f|)$. The number of edges in $G_{K,f}$, i.e. $|E| = O(|R||f| + |S||f|)$. It is not difficult to see that steps 1, 2, 4 and 5 can all be implemented in time linear in $(|V| + |E|)$.

Step 3 can be implemented using an algorithm of complexity $O(|f|(|V|+|E|))$. This algorithm works as follows: It first identifies all nodes that lie on unlabeled ν-cycles. This is done as follows.

For each ν-subformula g of f and in the increasing lengths of g, consider the restriction of $G_{K,f}$ to unlabeled nodes of the form (s, g') where g' is a subformula of g. For each strongly connected component C of the restricted graph, find the type of the longest subformula in any node of C; if this formula is a ν-formula then mark all the nodes of C as nodes lying on a ν-cycle.

In the full paper we will show that the above algorithm correctly identifies all the nodes that lie on unlabeled ν-cycles. After this, it is straightforward to find nodes that have unlabeld paths to such cycles. Each iteration of the above algorithm can be implemented using an algorithm of complexity $O(|V| + |E|)$. Hence, all the iterations together take time $O(|f|(|V| + |E|))$.

Thus, the overall complexity of the algorithm is $O(|f|(|V|+|E|))$. Substituting for $|V|$ and $|E|$ in terms of $|S|$ and $|R|$, we see that the overall complexity of the above algorithm is $O(|f|^2(|S| + |R|))$.

The above algorithm can be naturally be extended to the logic L_2 with the same complexity. We will present this in the full paper. Thus, model checking for L_2 can also be done in time $O(|f|^2(|S| + |R|))$.

We compare the expressive power of the logics to well known branching time temporal logics. Consider the branching time temporal logic CTL*. Let the ECTL* (given in [13]) denote the extended version of the logic CTL* where each path formula can be as expressive as ω-regular expressions.

Theorem 3.2 *The logic L_2 is as expressive as ECTL*.*

Proof We give an outline of the proof. First, we show that any ECTL* formula of the form $E(p)$ where E is the existential path quantifier and p is an ω-regular expression can be translated in to the logic L_1. To achieve this, we translate p into a Buchi automaton A on infinite strings. It is well known that linear μ-calculus, i.e. μ-calculus interpreted over infinite strings is as expressive as Buchi automaton on infinite strings. The logic L_1 is similar to the linear μ-calculus.

We can apply the same techniques used as in the case of linear μ-calculus, to show that for any Buchi automaton A we can obtain a formula f in L_1 such that f is satisfied at a state in a Kripke structure iff there exists an infinite path from that state that is accepted by A. Applying this translation inductively, we can show that any formula of the logic ECTL* can be translated in to the logic L_2.

To prove that every formula in L_2 can be translated into ECTL*, it is enough to show that L_1 can be translated into ECTL*. This translation can be applied inductively, to show that any formula of L_2 can also be translated into ECTL*. Given a formula f in L_1, we can construct a Buchi automaton A on infinite strings, and from this obtain an equivalent ECTL* formula of the form $E(p)$ where p is an ω-regular expression. ■

We can also use the following alternate approach for model-checking for formulas in L_1 which will have the same complexity. We briefly describe this method. The proof of Theorem 3.2 shows that for each formula f in L_1 there exists a formula of the form $E(p)$ in ECTL*, where p is a path formula, such that f is equivalent to $E(p)$. In fact, from f, we can construct a parity string automaton A_f (see the next section for the definition of parity automaton) with the following property: f is satisfied at a node s_0 in a Kripke structure iff there exists an infinite path from s_0 that is accepted by A_f. The number of states in the automaton A_f will be $O(|f|)$ and the number of sets in the accepting condition will be $O(|f|)$. Now to check if the formula f is satisfied at state s_0 of K, we simply consider K as a string automaton and construct the product automaton of A_f and K, and check for non-emptiness of this product automaton. The size of the product automaton ,which will also be a parity string automaton, will be $O((|S|+|R|)|f|)$, and the number of sets in the accepting condition will be $O(|f|)$ (in fact, the graph $G_{K,f}$ constructed in our algorithm is itself can be considered as the product automaton). Checking non-emptiness for parity string automaton can be done using a procedure, similar to the one used for step 3 of the previous algorithm, of complexity linear in the product of the size of the automaton and the number of sets in the acceptance condition. The complexity of this method will also be $O(|f|^2(|S| + |R|))$.

4 Relationship between Model Checking and Automata

In this section we explore the relationship between the model checking problem for μ-calculus and the emptiness problem for automata on infinite trees. More specifically, we show that the model checking problem for the complete logic μ-calculus is euquivalent under linear reductions to the emptiness problem of a particular type of automata on infinite trees, called parity automata. This shows that there is an efficient model checking algorithm for μ-calculus iff there is an efficient algorithm for checking emptiness of parity automata.

A corollary of the above result is that the model checking problem for formulas of μ-calculus which are of the form $\nu y(g)$ where g is in normal form and μ is the only fixed point operator appearing in g, is equivalent to the non-emptiness problem for Buchi tree automata

First, we define parity tree automata (introduced in [5, 9]). A parity tree automaton A on infinite binary tree is a 5-tuple $(\Sigma, Q, q_0, \delta, F)$ where Σ is the input alphabet, Q is the set of automaton states, q_0 is the initial states, $\delta : (Q \times \Sigma) \rightarrow 2^{Q\times Q}$ is the next move relation and $F = (F_0, F_1, ..., F_k)$ where $F_0, F_1, ..., F_k$ is a sequence of mutually disjoint subsets of Q. F is called the acceptance condition. Note that, for any $a \in \Sigma$ and $q \in Q$, $\delta(q, a)$ is a set of pairs of the form (q', q'') where q' and q'' are automaton states; Intuitively, if the automaton is in state q and reads input a in the current node then the state of the automaton on the left child is going to be q' and its state on the right child is going to be q''. We denote the infinite binary tree by the set $\{0, 1\}^*$. The empty string denotes the root of the tree and the two children of x are $x0$ and $x1$. An infinite path σ in the tree is an infinite sequence of nodes starting with root node and such that each succeeding node is a child of the preceding node.

An input to the automaton is a marked infinite binary tree which is a function $\tau : \{0, 1\}^* \rightarrow \Sigma$. A run of r of A on input τ is a function $r : \{0, 1\}^* \rightarrow Q$, associating a state of the automaton with each node of the tree, such that $r(\epsilon) = q_0$, and for any $x \in \{0, 1\}^*$ $(r(x0), r(x1)) \in \delta(r(x), \tau(x))$. For a run r and an infinite path $\sigma = \sigma_0, \sigma_1, ..., \sigma_i, ...$, we let $r(\sigma)$ denote the infinite sequence of automaton states $r(\sigma_0), ..., r(\sigma_i), ...$ The run r is accepting if for every infinite path σ, the following condition is satisfied: there exists an even number l, $0 \leq l \leq k$, such that some state in F_l appears infinitely often in $r(\sigma)$ and each of the states in the set $(\bigcup_{l<j\leq k} F_j)$ only appears finitely often in $r(\sigma)$. We say that the automaton A accepts an input τ iff there exists an accepting run r of the automaton on the input τ.

A Buchi automaton is a parity tree automaton in which the accepting condition F is of length one, i.e. it has only one set. Note that this definition is equivalent to the standard definition of Buchi tree automaton.

Theorem 4.1 *Given a kripke structure $K = (S, R, L)$ and any μ-calculus formula f and a state $s_0 \in S$, we can obtain a parity tree automaton A of size $O((|S| + |R|)|f|)$ in time $O((|S| + |R|)|f|)$ such that*

- *the number of sets in the acceptance condition of A is less than or equal to $altdepth(f) + 1$, and*
- *A accepts at least one input iff $K, s_0 \models f$.*

Proof We briefly sketch the proof here. Let f be the given μ-calculus formula and K be the given Kripke structure. First we construct the graph $G_{K,f} = (V, E)$ as given in the previous section. Recall that each node in V is of the form (s, h) where $s \in S$ and h is a subformula of f. The edge set E is as defined in the previous section. For example, when h is of the form $\texttt{[R]}h'$, then for each $(s, s') \in R$ there is an edge from (s, h) to (s, h') in E. We call a node (s, h) in V to be an $\wedge$-node if h is of the form $h_1 \wedge h_2$ or is of the form $\texttt{[R]}h_1$ and (s, h) has at least two successors (i.e. edges to two distinct nodes); we call it to be an atomic node if h is P or $\neg P$ for some atomic proposition P and all other nodes in V are called $\vee$-nodes. Note that if (s, h) is an $\vee$-node then h can be a variable, or a μ-subformula, or a ν-subformula, or a subformula of the form $\texttt{[R]}h'$ and (s, h) has only one successor. We make the following assumption. Any non-atomic node has at most two successors, i.e. two edges leaving it. If this condition is not satisfied, we can introduce new intermediate nodes and edges so that this property is satisfied; actually, for each node u with k successors, if $k > 2$ then we introduce $k-2$ additional new nodes and $k-2$ additional edges. As a consequence, the size of $G_{K,f} = |V| + |E|$ at most doubles. The type of a new node that is introduced in the previous step is same as that of u, i.e. it is a $\wedge$-node if u is a $\wedge$-node, etc. After this, each $\wedge$-node has exactly two successors, while a $\vee$-node has either one or two successors.

The automaton $A = (\Sigma, Q, q_0, \delta, F)$ is defined as follows. The states set Q of the automaton is simply V, the initial state q_0 is (s_0, f), the input alphabet Σ has only one symbol, say symbol a. The transitions of A are defined as follows. For any node $u = (s, h)$, $\delta(u, a)$ consists of the following pairs: if u is an atomic node (i.e. $h = P$ or $h = \neg P$) then $\delta(u, a) = \{(u, u)\}$; if h is $\vee$-node then $\delta(u, a) = \{(v, v) : (u, v) \in E\}$; if h is a $\wedge$-node then $\delta(u, a) = \{(v, v') : v$ and v' are the successors of $u\}$.

We define the acceptance condition $F = (F_0, F_1, ..., F_{(k-1)})$ as follows. Let T be the set of all of $g \in SF(f)$ such that g is μ-subformula or is a ν-subformula. First, arrange the members of T into a sequence so that for any two distinct h, g in T where h is a subformula of g, h appears before g. After this, group all subformulas of same type that appear contiguously in the above sequence into one set and obtain a sequence $C_0, C_1, ..., C_{(l-1)}$ of sets of subformulas that forms a partition of T satisfying the following properties:

- Each C_i contains subformulas of the same type, i.e. all of them are μ-subformulas or all of them are ν-subformulas. In addition, for each $i < (l-1)$, C_i and $C_{(i+1)}$ contain different types of formulas.
- For any $g, h \in T$ where g and h are of different type and g is a subformula of h, if g appears in some set C_i then h appears in a later set, i.e. h appears in C_j for some $j > i$.

We define $F = (F_0, F_1, ..., F_{(k-1)})$ according to the following three cases.

C_0 **contains ν-subformulas:** In this case $k = l$. $F_0 = (S \times C_0) \cup U$ where U is the set of all atomic nodes u such that u is of the form (s, P) and $P \in L(s)$, or is of the form $(s, \neg P)$ and $P \notin L(s)$. For $0 < i < k$, $F_i = S \times C_i$.

C_0 **contains μ-subformulas and $l > 1$:** In this case $k = l-1$. $F_0 = (S \times C_1) \cup U$ where U is same as in the previous case. For $0 < i < k$, $F_i = S \times C_{i+1}$.

C_0 **contains μ-subformulas and $l = 1$, or T is empty:** In this case $k = 1$. $F_0 = U$ where U is as in the previous cases.

The last of the above cases occurs if the formula has no νs. It can be shown that the automaton A accepts at least one input iff $K, s_0 \models f$. It is also not difficult to see that the size of the automaton, which is the sum of the number of states, the total number of transitions and the sum of the cardinalities of all sets in F, is $O((|S| + |R|)|f|)$. ∎

Theorem 4.2 *Given a parity tree automaton $A = (\Sigma, Q, q_0, \delta, F)$, we can obtain a Kripke structure K whose size is linear in the size of A and a μ-calculus formula f which is linear in the length of the acceptance condition, and a state s_0 in K, such that*

- *$altdepth(f) = 1+$ the number of sets in F, and*
- *A accepts at least one input iff $K, s_0 \models f$.*

Proof This theorem follows from the results of [5]. However, for the sake of completeness, we outline the major steps of the proof.

Without loss of generality, we can assume that the alphabet of A is a singleton consisting of the symbol a. Let the acceptance condition F be given by $(F_1, F_2, ..., F_{k-1})$ (note that index of the first set in F is 1). Let $F_0 = Q - \bigcup_{1 \leq i < k} F_i$. Now, consider the sets $F_0, F_1, ..., F_{k-1}$. Corresponding to each F_i, we use an atomic proposition P_i. We give the informal description of the Kripke structure $K = (S, R, L)$. S has the following elements. We call each element of S as a node and each element of R as an edge. Corresponding to each automaton state s, there is one node in S which is also denoted by s. For such a node s, if $s \in F_i$,for some $i < k$, then $L(s) = \{P_i\}$, otherwise $L(s)$ is the empty set.

Corresponding to each pair of states (s_1, s_2), such that $(s_1, s_2) \in \delta(s, a)$ for some state s, S has a node which we denote by the triple (s, s_1, s_2) and $L((s, s_1, s_2)) = L(s)$. R has the following edges. Corresponding to the above transition, there is an edge from the node s to the node (s, s_1, s_2) and there are edges from (s, s_1, s_2) to s_1 and to s_2. f is given by the following formula:
$\lambda_{k-1} x_{k-1} \lambda_{k-2} x_{k-2} ... \lambda_0 x_0 (\bigvee_{0 \leq i < k} P_i \wedge \texttt{<R>[R]} x_i)$

where $\lambda_{k-1} ... \lambda_0$ is an alternating sequence μs and νs ending with μ. Now, take $s_0 = q_0$. From the results of [5], it follows that $K, s_0 \models f$ iff A accepts at least one input. ∎

Corollary 4. *The model checking problem for formulas of the form $\nu x f$ where f is in normalized form and has no further νs appearing in it is equivalent to checking non-emptiness of Buchi tree automata.*

Theorem 4.3 *The model checking problem for the full μ-calculus is in NP∩co-NP. Formally, the set of encodings of all triples (K, s_0, f) satisfying the following condition is in NP∩co-NP: K is a Kripke structure, s_0 is a state in K and f is a μ-calculus formula such that $K, s_0 \models f$.*

5 Conclusion

In this paper we considered two different fragments of μ-calculus, logics L_1 and L_2. We gave model checking algorithms for logics L_1 and L_2 which are of complexity $O(m^2 n)$ where m is the length of the formula and n is the size of the structure. We have shown that the logic L_2 is as expressive as ECTL* given in [13]. In additions to these results, we have shown that the model checking problem for the μ-calculus is equivalent to the non-emptiness problem of parity tree automata.

It will be interesting to investigate if there is a model checking algorithm for the logics L_1 and L_2 which is only of complexity $O(mn)$ instead of $O(m^2n)$. Of course, determining if the model checking problem for the full μ-calculus is in P or not, is also an open problem.

References

1. R. Cleaveland, *Tableux-based model checking in the propositional μ-calculus* , Acta Informatica, 27:725-747, 1990.
2. R. Cleaveland and B. Steffen, *A linear-time model-checking for alternation free modal μ-calculus* , Proceedings of the 3rd workshop on Computer Aided Verification, Aalborg, LNCS, Springer-Verlag, July 1991.
3. R. Cleaveland and B. Steffen, *Faster model-checking for modal μ-calculus* , Proceedings of the 4th workshop on Computer Aided Verification, Montreal, July 1991.
4. E. A. Emerson, E. M. Clarke, *Characterizing correctness properties of parallel programs Using Fixpoints*, Proceedings of the International Conference on Automata, Languages and Programming, 1980.
5. E.A. Emerson and C. S. Jutla, *Tree Automata, Mu-calculus and Determinacy*, Proceedings of the 1991 IEEE Symposium on Foundations of Computer Science.
6. E. A. Emerson and C. Leis, *Efficient model-checking in fragments of μ-calculus* , Proceedings of Symposium on Logic in Computer Science, 1986.
7. E. A. Emerson and C. Leis, *Modalities for Model Checking*, Science of Computer Programming, 1987.
8. D. Kozen, *Results on the propositional μ-calculus*, Theoretical Computer Science, 27, 1983.
9. A.W. Mostowski, *Regular Expressions for Infinite trees and a standard form of automata*, in: A. Skowron, ed., *Computation Theory*, LNCS, vol 208, 1984, Springer-Verlag.
10. D. Niwinski, *Fixed-points Vs. Infinite Generation*, Proceedings of the Third IEEE Symposium on Logic in Computer Science, 1988.
11. C. Stirling, D. Walker, *Local model-checking in modal μ-calculus*, Proceedings of TAPSOFT, 1989.
12. R. S. Streett and E. A. Emerson, *An automata theoretic decision procedure for Propositional μ-calculus* , Proceedings of the International Conference on Automata, Languages and Programming, 1984.
13. M. Vardi and P. Wolper, *Yet Another Process Logic*, Proceedings of the workshop on Logics of Programs, Pittsburgh, 1983, also appeared in Lecture Notes in Computer Science.

This article was processed using the LaTeX macro package with LLNCS style