Linear weak alternating automata and the model checking problem

Moritz Hammer

May 13, 2004

Abstract

Abstract. Automata-based model checking is a widely used approach towards software model checking. Traditionally, nondeterministic Büchi automata are used to represent the temporal logic property to be checked. We take a look at a special kind of alternating automata, the linear weak alternating automata. They can be constructed from LTL formula in an elegant way in linear time. The emptiness check on linear weak alternating automata, on the other hand, requires exponential time, whereas the emptiness of nondeterministic Büchi automata, being of exponential size with respect to the size of the LTL formula they represent, can be checked in linear time. We try to use the advantage of the better constructability of linear weak alternating automata in model checking by implementing a model checker using "on-the-fly generalized Büchi automata generation". The emptiness check is conducted using an extended version of Tarjan's algorithm. After obtaining promising results, we adapt the SPIN model checker to the new algorithm.

1 Introduction

1.1 The model checking problem

Model checking of finite state systems has been popular for many years, as it can provide some significant advantages over other verification techniques: first, it is completely automatic, requiring just the input of the model and a temporal-logic formula. It is a "push-button-approach", freeing the user of the need to follow the actual proof, as required by a semi-authomatic theorem prover. Second, it conducts an exhaustive search and can prove a model to be correct. Third, it can display a counterexample if an error is found which can be used to understand the error and subsequently change the model.

Model checking is called "semantic verification", indicating that the semantics of property and model are examined to validate their conformance. To put it simple, model checking just checks if every temporal structure that is allowed by the model satisfies the property too. However, the complexity of LTL-model checking is known to be EXPTIME-complete (actually, linear with respect to the model size, but exponential with respect to the formula), making model checking challenging in both runtime and memory requirements. In practice, runtime requirements can make model checking infeasible, however memory requirements can make it impossible. Therefore, a good model checker will try to save memory as well as try to be fast.

1.2 Approaches to the model checking problem

Model checker have been used for both hardware and software verification. Hardware verification is concerned with proving the correctness of circuits to avoid undetected flaws like the faulty FDIV unit that led to the well-remembered "Pentium bug". As these circuits are designed using CAD systems, the model to check can be taken from these designs directly. Software verification, however, usually uses hand-written models of the actual algorithm to check. It is not entirely impossible to check software directly, however with current computers and algorithms, it seems quite futile to attempt this without quite excessive automatic abstraction.

For software model checking automata-based approaches are widely used. Because these algorithms search for possible property violations by looking at all possible paths, they are known as explicit model checker.

Today, SPIN is the most famous (freely available) member of this model checker class (see [Hol97, Hol03]). In hardware verification, symbolic model checking [McM92] has gained much acceptance. Using BDDs for state representation, the memory requirements were decreased. As hardware verification produces models with many variables, yet usually little trace length, the symbolic model checking approach produced good results. Another symbolic technique is known as bounded model checking [BCRZ99], which uses propositional logic formulae for fault detection. As it relies on the SAT-problem (see [Coo71] for Cook's famous description of the problem), it can make use of highly optimized SAT solvers. Bounded model checking does not search for all possible traces at once, instead it just looks for those shorter than a given bound. This enables the verification of short traces for very large models, but makes it difficult to prove a model correct. Recently it has been shown by Clarke et al [CKSO04] that the worst case trace length to verify is exponential in the size of the model. However, hardware model checking is concerned with finding errors rather than proving correctness, thus bounded model checking received a lot of attention.

In this thesis we shall be concerned with software model checking. Compared to hardware model checking, the models usually feature less variables, the traces tend to get longer and correctness proofs are desired. Consider, for example, a mutual exclusion algorithm (like Peterson's algorithm). The model consists of maybe four bit variables and two program counters, but if we want to prove "single-bounded overtaking", counterexample traces can get quite long. We also want to show the correctness of the algorithm, as it is pretty likely that if there exists a possibility of a flawed run, it will occur eventually.

1.3 Thesis overview

In the first section, the theoretical background is presented, which is concerned with different kinds of omega automata and their language emptiness checks. We then discuss an implementation of a linear weak alternating automata-based model checker, and present some benchmarking results. After that, we outline the adaption of the SPIN model checker to use the new algorithm.

This paper is a shortened preliminary result presentation. It omits the reimplementation of the Gastin/Oddoux algorithm to generate nondeterministic Büchi automata, as well as various (not very successful) attempts to minimize the linear weak alternating automata. It also does not contain data for the *lwaaspin* implementation, which is not yet ready to be tested against large models at the time of writing.

1.4 Related work

This thesis follows the paper of Merz/Sezgin [MS03] and Gastin/Oddoux [GO01]. Gastin/Oddoux present *ltl2ba*, a very efficient tool to convert LTL formulae to nondeterministic Büchi automata. This approach has been refined by Fritz [Fri03] and Tauriainen [Tau03b]. This way is the "state-of-the-art" approach towards nondeterministic Büchi automata generation, but we shall only briefly discuss it in this paper.

In the main part we present the implementation of an emptiness check on linear weak alternating automata based on Merz/Sezgin's algorithm. As far as we know this has not been done before. Our algorithm performs a combined on-the-fly generation of a generalized Büchi automaton and an emptiness check on this automaton. An emptiness check based on the "CVWY"-algorithm [CVWY92] for generalized Büchi automata has been proposed by Tauriainen [Tau03a], but our algorithm is an adaptation of the strongly connected component detection algorithm described by Tarjan [Tar72].

We use this algorithm as a model checker and add it to SPIN [Hol97, Hol03].

2 Theory

2.1 Automata-based model checking

The task of model checking is to see whether all runs allowed by a model of a system are also allowed by a property. For example, one might wonder whether a given model of a counter will always reset to zero after a finite number of steps. As the model is declared as a state transition system it is pretty easy to see if a run is allowed by the model. However, it is not so easy to see the compliance of a run with a property given by a temporal logic formula. Automata-based model checking, introduced by Vardi and Wolper (see [VW86]), deals with this problem by using a special kind of automata.

Let Σ be an alphabet. As a run is an infinite word (denoted $w \in \Sigma^\omega$), the automata used have to provide acceptance on infinite words. Most commonly, nondeterministic Büchi-automata have been used for this. They look almost like a common NFA, but offer a different acceptance condition:

Definition 1 (Nondeterministic Büchi Automaton). *A nondeterministic Büchi automaton is a tuple* $\mathcal{B} = (Q, q_0, \delta, F)$ *where*

- Q *is a finite set of locations*
- $q_0 \in Q$ *is the initial location*
- $\delta \subseteq Q \times \Sigma \times Q$ *is the transition relation*
- $F \subseteq Q$ *is the set of accepting locations*

For a infinite word $w = w_0 w_1 \ldots \in \Sigma^\omega$ *the automaton* $\mathcal{B}$ *is accepting iff there is a path* $\rho = p_0, p_1, \ldots$ *of configurations with* $p_0 = q_0$, $(p_i, w_i, p_{i+1}) \in \delta$ *for all* $i \in \mathbb{N}$ *and* $p_i \in F$ *for infinitely many* i.
The language $\mathcal{L}(\mathcal{B})$ *is defined as* $\mathcal{L}(\mathcal{B}) = \{w \in \Sigma^\omega \mid \mathcal{B} \text{ accepts } w\}$.

Unlike finite automata, nondeterministic Büchi automata are more expressive than deterministic ones. For example, the property $\Diamond\Box f$ cannot be expressed by a deterministic automaton (the informal proof idea is the lack of "guessing" capability of the point where no $\neg f$ will occur anymore). However, nondeterminism and alternation are equal for automata on infinite words - they both recognize the omega-regular languages. For finite words, all three automata types are equivalent to regular languages.
How can ω-automata be used for model checking purposes? Given a model M and a LTL formula φ, we want to show that every run allowed by the model is allowed by the property φ, too:

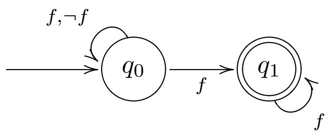


Figure 1: Nondeterministic Büchi automaton for $\Diamond\Box f$

$$M \models \varphi$$

To achieve this, we might generate the automaton $\mathcal{A}_\varphi$ and check if its language is a superset of the language of the model M:

$$\mathcal{L}(M) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$$

This is equivalent to showing that no run of the automaton satisfies the inverted formula:

$$\mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi}) = \emptyset$$

To calculate the intersection, we can use the product automaton:

$$\mathcal{L}(M \times \mathcal{A}_{\neg\varphi}) = \emptyset$$

Thus, automata-based model checking involves two steps: First, building the product automaton, and second, checking if its language is empty. If it is, our property holds in the model. If it is not, we can find an infinite word that will provide an example of a run that is valid in the model, but invalid with respect to the property. However, in practice the product automaton is far too big to be calculated in a distinct step. Therefore, on-the-fly model checking is conducted by most automata-based model checkers: both steps are combined and the automaton locations are calculated just when needed. This technique is the foundation of the most popular LTL model checker, SPIN [Hol97].

2.2 Alternating automata

We are interested in other types of automata over infinite words for two reasons: First, nondeterministic Büchi automata recognize the omega-regular languages, which are the languages describeable by the μ-calculus. As LTL is equivalent to the omega-starfree languages and thus a proper subset of the languages

recognized by nondeterministic Büchi automata (and many other automata like Muller-, Rabin- and Streett-automata), nondeterministic Büchi automata are too strong for our purposes. This does not yield a problem, however we might be faster or more memory efficient with weaker automata.
Second, nondeterministic Büchi automata have exponential size with respect to the LTL formula they represent. For simple properties this is no problem, as a human-formulated LTL property will not be very big. However, one of the advantages of LTL (over the computation tree logic, CTL) is the ability to add fairness constraints to the property, but this will result in larger formulas. Alternating automata do have the advantage of being of linear size with respect to the formula. To determine whether this fact can be used to formulate a more efficient model checking algorithm will be topic of this thesis.
Alternating automata add universal choice to nondeterministic automata. This makes them logarithmical smaller than nondeterministic Büchi automata accepting the same language, but it also makes them far more difficult to handle. Thus we might want to trade power of expression for easier handling: as it was shown by Thomas [Tho97], a subclass of alternating automata is sufficient for LTL: the linear weak alternating automata[1].

Let us first introduce alternating automata as proposed by Muller and Schupp [MS84, MS87]:

Definition 2 (Alternating automata). *A alternating automaton is a tuple $\mathcal{A} = (\Sigma, Q, \Delta, q_0, F)$ where*

- *Σ is the alphabet*
- *Q is the set of locations*
- *$\Delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the transition function, where $\mathcal{B}^+(Q)$ is a boolean formula containing only positive variables $q \in Q$.*
- *$q_0 \in Q$ is the initial location*
- *$F \subseteq Q$ is the set of final locations (either co-Büchi [GO01], or Büchi [FW03, FS01])*

Because of their Büchi (or co-Büchi, where each final state may be visited only finitely often in an accepting run) acceptance, these automata are also known as alternating Büchi automata.
To make this alternating automaton linear weak, we propose a restriction on the transition function:

Definition 3 (Linear weak alternating automata). *Let $\mathcal{A}$ be an alternating automaton. $\mathcal{A}$ is called a linear weak alternating automaton iff there exists a partial order on Q such that for all $q \in Q$, all the locations appearing in $\delta(q)$ are lower or equal to q.*

For practical reasons, it is feasible to use another presentation of the transition function $\delta : Q \to \mathcal{B}(Q \cup \Sigma)$ which is equivalent but more easily manageable. We also would like to use another formulation of the acceptance condition, which is also equivalent, but can be derived from the LTL formula more easily. Thus, following the definition of Merz and Sezgin [MS03], we give another definition of linear weak alternating automata:

Definition 4 (Linear weak alternating automata, alternative definition). *A linear weak alternating automaton is a tuple $\mathcal{A} = (\mathcal{V}, Q, q_0, \delta, \rho)$ where*

- *$\mathcal{V}$ is the set of atomic propositions*
- *Q is the set of locations*
- *$q_0 \in Q$ is the initial location*
- *$\delta : Q \to B(Q \cup \mathcal{V})$ is the transition function that assigns a boolean formula to each $q \in Q$; this formula must not contain negative $q' \in Q$*
- *$\rho : Q \to \mathbb{N}$ is a ranking function*

[1] also known as “very weak alternating automata”

To make this a linear weak alternating automaton, it is required that $\forall q, q' \in Q . q \rightarrow^* q' \wedge q' \rightarrow^* q \Rightarrow q = q'$.

As opposed to the first definition, the acceptance condition is not a Büchi condition but uses the ranking function. The basic idea here is that even-ranked locations are accepting, while odd-ranked locations are rejecting. This is why these automata are also known as “linear parity alternating automata”.
We will use the second definition for linear weak alternating automata during this thesis. To define acceptance, we first have to introduce run dags (directed acyclic graphes) and paths:

Definition 5 (Run dags and paths of linear weak alternating automaton). *Let* $\mathcal{A} = (\mathcal{V}, Q, q_0, \delta, \rho)$ *be a linear weak alternating automata and* $\sigma = s_0 s_1 \ldots$ *be a sequence of inputs* $s_i \subseteq \mathcal{V}$*. A run dag of* $\mathcal{A}$ *over* σ *is represented by the* ω*-sequence* $\Delta = e_0 e_1 \ldots$ *of its edges* $e_i \subseteq Q \times Q$*. The configurations* $c_0 c_1 \ldots$ *of* Δ *are defined by* $c_0 = \{q_0\}$ *and* $c_{i+1} = ran(e_i)$ *with* $ran(r)$ *being the range of the relation* r*. We require that for all* $i \in \mathbb{N}$*,* $dom(e_i) \subseteq c_i$ *and that for all* $q \in c_i$*,* $s_i \cup e_i(q)$ *satisfies* $\delta(q)$*.*
A path in a run dag Δ *is a maximal sequence* $\pi = p_0 p_1 \ldots$ *of locations* $p_i \in Q$ *such that* $p_i \in c_i$ *and* $(p_i, p_{i+1}) \in e_i$ *for all* i *such that* p_i *(respectively* p_{i+1}*) appears in* π*.*

A run dag is a graph of activated locations over some infinite input word. It is clear that the breadth of this dag cannot exceed the number of locations. To avoid confusion, this dag represents just a single path through the existential choices of the alternating automaton. The branches it takes resemble the requirements of universal choice. Paths are only of theoretical interest here, as they do not resemble anything practical. However, we need them to define the acceptance condition of linear weak alternating automata:

Definition 6 (Acceptance of linear weak alternating automata). *Let* $\mathcal{A} = (\Sigma, Q, q_0, \delta, \rho)$ *be a linear weak alternating automata. A run dag* Δ *of* $\mathcal{A}$ *is accepting iff* $\min(\rho(p_i) \mid i \in \mathbb{N})$ *is even for every infinite path* $\pi = p_0 p_1 \ldots$*.*
Let $w \in (2^{\mathcal{V}})^{\omega}$ *be an infinite word.* $\mathcal{A}$ *accepts* w *iff there exists a run dag* Δ *over* w *that is accepting.*

The dag of the linear weak alternating automaton has the property that for some ordering of the locations (as given by the ranking function), there is no rising edge. Thus, the ranks occurring along any path will be monotonically decreasing, and as there are only finite many ranks, it will eventually get trapped in one. The basic idea of the emptiness check is to ensure that every path will get trapped in an even-ranked location, as we will see later.

2.3 Transformation of LTL formulae into linear weak alternating automata

Translating a LTL formula into a linear weak alternating automaton is rather straight forward. LTL was introduced by Pnueli in 1977 (see [Pnu77]); we shall omit the formal description of syntax and semantics of LTL in this paper.

To build a LWAA for a LTL formula, we assume the formula to be given in negation normal form (NNF). In NNF, negations must not be placed anywhere else but in front of atomic propositions. The automaton $\mathcal{A}_\varphi$ for the LTL property φ is given as $\mathcal{A}_\varphi = (\mathcal{V}, Q, q_\varphi, \delta, \rho)$ as follows:

- $\mathcal{V}$ is the set of propositions
- Q contains a location for every subformula of φ
- q_φ ist the location representing φ
- δ and ρ are given by the following table, where $\lceil n \rceil_{\text{odd}}$ (resp. $\lceil n \rceil_{\text{even}}$) denotes the smallest odd (resp. even) number that is at least n:

location q	$\delta(q)$	$\rho(q)$
q_φ (φ non-temporal)	φ	0
$q_{\varphi\wedge\psi}$	$\delta(q_\varphi) \wedge \delta(q_\psi)$	$\max\{\rho(q_\varphi, q_\psi)\}$
$q_{\varphi\vee\psi}$	$\delta(q_\varphi) \vee \delta(q_\psi)$	$\max\{\rho(q_\varphi, q_\psi)\}$
$q_{\circ\varphi}$	q_φ	$\rho(q_\varphi)$
$q_{\varphi \text{ until } \psi}$	$\delta(q_\psi) \vee (\delta(q_\varphi) \wedge q_{\varphi \text{ until } \psi})$	$\lceil\max\{\rho(q_\varphi, q_\psi)\}\rceil_{\text{odd}}$
$q_{\varphi \text{ release } \psi}$	$(\delta(q_\psi) \wedge \delta(q_\varphi)) \vee (\delta(q_\psi) \wedge q_{\varphi \text{ release } \psi}))$	$\lceil\max\{\rho(q_\varphi, q_\psi)\}\rceil_{\text{even}}$
$q_{\Box\varphi}$	$\delta(q_\varphi) \wedge q_{\Box\varphi}$	$\lceil\rho(q_\varphi)\rceil_{\text{even}}$
$q_{\Diamond\varphi}$	$\delta(q_\varphi) \vee q_{\Diamond\varphi}$	$\lceil\rho(q_\varphi)\rceil_{\text{odd}}$

It is quite obvious that the resulting automaton is indeed linear weak. The correctness of this translation has been shown by Vardi [Var95] (see also [Mer00]).

The translation decomposes the temporal operators according to their fixpoint characterizations. $\Box\varphi$ corresponds to a greatest fixpoint, therefore it must be of even (accepting) rank. The corresponding location "copies" the transitions of the location of φ and adds a "self-link" to each of them. Thus, once the $\Box\varphi$-location is activated, it will keep itself activated forever.

2.4 Deciding emptiness on omega automata

2.4.1 Nondeterministic Büchi automata

We first give a brief introduction to the Courcoubetis, Vardi, Wolper and Yannakakis "nested depth first search" algorithm (CVWY algorithm), which is the foundation of the algorithm used in SPIN.

```
void dfs(boolean cycleflag) {
  State s = stack.top();
  if (!visited.contains(s, cycleflag)) {
    visited.insert(s, cycleflag);
    if (cycleflag && s == seed) {
      throw new TracefoundException(seed, stack);
    }
    foreach (t in successors(s)) {
      stack.push(t);
      dfs(cycleflag);
    }
    if (!cycleflag && s.isAcceptanceState()) {
      seed = s;
      foreach (t in successors(s)) {
        stack.push(t);
        dfs(true);
      }
    }
  }
  stack.pop();
}


main() {
  visited = new Set();
  stack = new Stack();
  seed = null;
  foreach (t in initialStates) {
    stack.push(t);
    dfs(false);
  }
}
```

Figure 2: Pseudo-Code of the CVWY algorithm

The algorithm searches for nontrivial cycles in linear time in time linear in the product of the model and the Buchi automaton. The basic idea is to perform a nested depth-first-search: the algorithm first traverses the tree, and if a acceptance state is found, it attempts to find a cycle. As the tree gets traversed first, and the search for cycles is conducted later, there is no need to check an already visited state (visited in a cycle search) again, as if it would have been in a cycle, the cycle would have been found already. Thus, each state is visited at most twice, providing linear complexity. In addition, there are some possible optimizations to the algorithm: the state-map, which stores each state seen, can be compressed by lossless and lossy compression.

Lossy compression, called "bit-state hashing" (see [Hol03] for details), relies on using a bit-hashmap to identify visited states. Instead of storing each state in a hashtable, only a bit is set if a state with a given hashcode is visited. Of course, the completeness of the algorithm voids, but soundness is preserved. What is more, it can be estimated how likely a hash collision is, providing an easy approach towards probabilistic model checking, which allows for much larger models.

The simplicity and good performance of the CVWY algorithm make it a hard benchmark for other emptiness checks. However, there is another algorithm, proposed by Tarjan [Tar72] and evaluated again only recently by Geldenhuys and Valmari [GV04] with surprisingly good results. This algorithm, called Tarjan's algorithm, is intended to identify strongly connected components (SCCs) in a directed graph. A strongly connected component C of a graph $G = (V, E)$ is a maximum subset of the node set ($C \subseteq V$), which satisfies that all nodes $n \in C$ are pairwise connected. It is important to note that because the subset is maximal, all SCCs of a graph G are distinct. The nodes not inside an SCC are "one-way nodes" because in any infinite path of the graph, they can be visited once at most.

The idea of the algorithm (see [NSS94] for a good presentation) is to maintain a stack containing all SCC member node candidates and identify the root candidate, which is the earliest node of the stack found to be inside the SCC. If an earlier node is found to be strongly connected (by being reachable from the SCC as well as being on the stack, which ensures it can in turn reach the SCC), it is made the root candidate. Once all nodes have been evaluated, all the nodes down to the root candidate can be removed from the stack, as they are known to be inside a SCC. Let `min(a,b)` return the stack element out of $\{a, b\}$ closer to the stack bottom in the following algorithm:

```
void visit(Node n) {
   push(n);
   root[n] = n;
   inComp[n] = false;
   for each successor s of n {
      if (s not visited) visit(s);
      if (not inComp[s]) root[n] = min(root[n], root[s]);
   }
   if (root[n] == n) {
      while (top() != n) {
         pop();
      }
   }
}

void main() {
   stack = empty;
   for each node n {
      if (n not visited) visit(n);
   }
}
```

Figure 3: Pseudo-Code of Tarjans algorithm

The runtime is $O(|V| + |E|)$, as all nodes will be visited once, and each outgoing edge will be examined. The most interesting thing about that algorithm is that it features two stacks: a recursion stack, working as in the CVWY algorithm, and a SCC candidate stack.

To use this algorithm for model checking, we can search for SCCs that contain all the acceptance conditions that need to be satisfied. We can just write them into the root candidate node, and transfer them if the root node gets predated. This works with a single acceptance condition (like with a nondeterministic Büchi automaton) as well as with a set of acceptance conditions, like those used with generalized Büchi automata.

2.4.2 Linear weak alternating automata

Usually, linear weak alternating automata are translated into nondeterministic Büchi automata to perform emptiness checks on them. We may, however, tackle the automaton directly by inspecting the configuration graph. The configuration graph is nothing else but a generalized Büchi automaton (generalized, because there are sets of acceptance conditions instead of a single condition). It contains a node for each element of the powerset of the linear weak alternating automaton locations, and transitions between nodes where a configuration change is possible. This is exactly the same schema as known from the conversion algorithm of nondeterministic finite automata to deterministic finite ones. It also yields the same exponential node blowup.

We repeat the central theorem and proof of the work of Merz/Sezgin [MS03], as it will be the basis of our model checking algorithm:

Let $G_{\mathcal{A}} = (V, E, \lambda)$ be the configuration graph of $\mathcal{A}$, with $V \subseteq 2^Q$ being the set of configurations and E containing an edge $(c, c') \in 2^Q \times 2^Q$ iff for some input state s, c' is minimal successor configuration of c,

```
void visit(Node n) {
   push(n);
   root[n] = n;
   inComp[n] = false;
   for each successor s of n {
      if (s not visited) visit(s);
      if (not inComp[s]) {
         if (root[s] < root[n]) {
            accset[root[s]] = accset[root[s]] + accset[root[n]];
            root[n] = root[s];
         }
         accset[root[s]] += (acceptance conditions of s);
         if (accset[root[s]] satisfies all acceptance conditions) {
            exit("cycle found");
         }
      }
   }
   if (root[n] == n) {
      while (top() != n) {
         pop();
      }
   }
}

void main() {
   stack = empty;
   for each node n {
      if (n not visited) visit(n);
   }
}
```

Figure 4: Pseudo-Code of Tarjans algorithm with acceptance set enhancement

i.e. $\forall q \in c.s \cup c' \models \delta(q)$ while this holds for no $c'' \subsetneq c'$. $\lambda : E \to 2^{Q_{\text{odd}}}$ is an edge labeling function that assigns to every edge the set of odd-ranking locations $q \in Q_{\text{odd}}$ such that no transition of $\mathcal{A}$ from c to c' can avoid performing a self-loop at q, i.e. $q \in e(q)$ holds for all transitions e from c to c', for all possible states $s \in 2^{\mathcal{V}}$.

Theorem 1. *Let $\mathcal{A}$ be a linear weak alternating automaton. Then $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $G_{\mathcal{A}}$ contains a nontrivial strongly connected component C reachable from configuration $\{q_0\}$ such that the intersection of $\lambda(c, c')$, for all edges $(c, c') \in C$, is empty.*

Proof. "Only if": If $\mathcal{L}(\mathcal{A}) \neq \emptyset$, then $\mathcal{A}$ admits a finite dag Δ satisying the conditions of the emptiness condition on linear weak alternating automata given above.. Consider the (nontrivial) SCC of $G_{\mathcal{A}}$ containing the configurations $c_k, \ldots, c_n$ of the loop of Δ, which is clearly reachable from $c_0 = \{q_0\}$. Because for every $q \in Q_{\text{odd}}$, we have $q \notin e_j(q)$ for some $k \leq j \leq n$, we find that $q \notin \lambda(c_j, c_{j+1})$ and thus the intersection of all sets $\lambda(c, c')$ is empty.
"If": Assume given a nontrivial SCC of $G_{\mathcal{A}}$ that is reachable from $\{q_0\}$ and such that the intersection of all sets $\lambda(c, c')$, for edges (c, c') between configurations in C is empty. We can construct a finite dag as follows: first, construct a dag from $\{q_0\}$ to the root of C from $G_{\mathcal{A}}$. Second, for every $q \in Q_{\text{odd}}$, C must contain some transition (c, c') such that $q \notin \lambda(c, c')$, hence there must be some transition e (for some state s) from c to c' such that $q \notin e(q)$. Because C is a SCC, we can construct a finite path within C that contains all these transitions for every $q \in Q_{\text{odd}}$. □

If we calculated the configuration graph in advance, we would get an ordinary generalized Büchi automaton, exponentially sized with respect to the original linear weak alternating automaton. In fact, the algorithm proposed by Gastin/Oddoux does exactly this. However, we will attempt to calculate the configuration graph on the fly. As we will obtain a generalized Büchi automaton, we cannot use the CVWY algorithm and have to use a modified version of Tarjan's algorithm instead.

2.5 Complexity issues

As LTL model checking has been shown to be EXPTIME-complete (see [May98], see also [SC85] for a proof of PSPACE-completeness of LTL satisfiability checking), there is no hope of getting rid of the exponential blowup at some point of the algorithm. Using alternating automata seems feasible because it postpones the exponential blowup to the emptiness check, whereas using nondeterministic Büchi automata provides a linear-time emptiness check, but only on the exponential-sized automaton. In theory, the nondeterministic Büchi automata approach performs two steps, each with exponential time in the size of the LTL formula (worst-case). The alternating automata approach just features one exponential timed step, plus a linear step that should be of no concern.

Thus, we can hope for a few improvements by using alternating automata:

$$\begin{tikzcd}
 & BA \arrow[r, "linear"'] & \text{Emptiness Decision} \\
LTL \arrow[ur, "exponential"'] \arrow[r, "linear"'] & LWAA \arrow[ur, "exponential"'] &
\end{tikzcd}$$

Figure 5: Transformation complexity

First, there should be no real restriction on the size of the formula to be checked. In almost any given scenario, it should be possible to start the emptiness test, making it possible to use abstraction techniques there. If the formula gets too large, these techniques fail in SPIN, as it fails to complete the generation of the automaton. It could be possible that the linear-sized automaton translation, combined with super-state hashing, can provide a probabilistic model checking algorithm capable of at least attempting to solve much larger formulas on large models. However, if super-state hashing can be applied to the linear weak alternating automaton emptiness check remains to be seen.

Second, there is hope that the overall process could be faster because of the lesser time spent on the automaton generation.

Third, alternating automata might offer some approaches for taking advantage of heuristics, as the accepting run dag is very intuitive; it might be possible to take advantage of that - which could improve fault detection, but not correctness proofs.

However, as Holzmann [Hol03] reported, the worst-case time (and space) requirement of nondeterministic Büchi automata is rarely seen in practice, and thus predictions are difficult to make. Whether the smaller size of the linear weak alternating automaton can compensate for the extended complexity of the emptiness check remains to be seen.

3 Implementation

3.1 Visiting the configuration graph

In an early attempt to familiarize ourselves with linear weak alternating automata, we reimplemented the algorithm proposed by Gastin/Oddoux [GO01]. We recognized that almost all of the runtime went into the conversion of the linear weak alternating automata to generalized Büchi automata, while the linear weak alternating automata itself was obtained almost instantly. Even for formula which can no longer be processed even by *ltl2ba* (in reasonable time, say a few hours), our Java implementation obtained the linear weak alternating automaton in less than a second.

Thus, we can start the emptiness check on linear weak alternating automata right away - trivial counterexamples can be found fast, and with the use of probabilistic algorithms like superstate hashing we might tackle much larger problems. However, as our algorithm is more complex, it might be slower in the average case; the exact behaviour remains to be seen in experiments on actual model checking problems.

In the modified Tarjan's algorithm, we perform on-the-fly-generalized Büchi automaton generation and emptiness checking together. Tarjan's depth-first-search has been modified to find cycles satisfying a number of conditions - in our case, the odd-ranked locations that have been found to be "not-self-activating" in the loop. In a nutshell, the algorithm identifies root candidates of strongly connected components andstores the labels in this root candidates. If the strongly connected component is found to be bigger, the root candidate is pushed further down the stack. Once all labels have been cleared, we have found an accepting cycle. If we completed identifying the strongly connected component without finding an accepting cycle, we can mark all nodes contained to be searched, they cannot show up in any further strongly connected component (as the nodes of two differend strongly connected components are necessarily disjunct).

The complexity of this algorithm is still $O(n + e)$, as it still visits every node just once and then pursues all outgoing edges once. However, the algorithm is more complicated to implement.

3.2 Extension to the model checking problem

Of course, an implementation of this algorithm for LTL satisfiability checking outperforms a Büchi-automata-based model checker pretty much on most examples, as most LTL formulas have rather short satisfiability examples (unless you use $\bigcirc\varphi$ a lot), and the search will often succeed much faster than the generation of the Büchi automata lasted. For a practical evaluation of the algorithm, we have to use it in a model checker. This makes counterexamples more difficult to find, as well as nontrivial unsatisfiable formulae easier to construct.
Extending the algorithm given above to the model checking problem is rather easy: A state of the product automaton consists of

- a configuration graph node $c \in 2^Q$ (which is the set of the linear weak alternating automata locations that are activated together) like we used in the satisfiability check
- a system state $v \in \Sigma = 2^V$ that describes the current state of the model

Unlike with satisfiability checking (where any successor configuration will do), we have to restrict the automaton successor configuration to those states that can be reached by transitions whose guards are enabled by the system state v. Likewise, only those system state successors that can be obtained from taking an enabled action are valid successor system states.
In a first, Java-based approach, the configuration graph successor of a configuration graph node $c \in 2^Q$ in the system state $v \in 2^V$ got calculated by combining all the transitions of the LWAA states contained in the configuration graph state:

$$\delta_{CG}(c) = \{c' \in 2^Q \mid \forall q \in c.c' \models \delta_{LWAA}(q, v)\}$$

This can be obtained by using all the combinations of the transitions offered by the LWAA states of q. This approach is quite fast, however it results in quite a lot of redundant transitions. Therefore, on-the-fly pruning is necessary to avoid visiting the same successor configuration multiple times.
The Java version was compared to another Java model checker that uses the Büchi automata translation described above. On unsatisfiable formulas, the linear weak alternating automata-based approach performed about three times slower than the "traditional" Büchi automata-based approach. On satisfiable formulae, it outperformed the traditional approach, which was encouraging enough to compare the new approach to SPIN.
To be comparable with SPIN, the next version was written in C++ and relied heavily on efficient data structures. This version also takes advantage of a very elegant approach towards the calculation of configuration graph successor nodes of the configuration c in system state v as proposed by Merz/Sezgin: Using BDDs to represent propositional formulae, we create the simple formula

$$v \wedge \left(\bigwedge_{q \in c} \delta(q) \right)$$

and retrieve all satisfying valuations from the representing BDD. This offers two advantages: first, the successor nodes found are distinct, and second, the set of successor nodes is minimal.
We also tried to use the BDD package to calculate the system successor states, but it turned out to be slower: the larger amount of variables used in the BDD package slowed things down. We thus reverted to calculating the system successor by applying the enabled actions and combining it with every configuration graph node. This also allowed for another improvement: Caching of the configuration graph successor calculation BDDs. Once a configuration graph node has been visited, we try to save the successor BDD in a hashtable cache. This avoids the expensive BDD rebuilding; however, the conjunction with the system state and thus the calculation of the satisfying valuations cannot be avoided. In our current implementation this is still the slowest part of the algorithm, and the most memory consuming too.
When comparing our implementation to SPIN using an unsatisfiable formula, SPIN usually outperforms our implementation by a factor of 6. This is neither unexpected nor discouraging, as SPIN is heavily optimized

and uses code generation to create a special model checker for each model and never-claim to check. On satisfiable formulae, our implementation is sometimes slower, but sometimes even faster. As both SPIN and our implementation use undirected search, this does not indicate much. However, if we check large, satisfiable LTL formulae on rather small models (due to memory constraints), our implementation presents a counterexample right before *ltl2ba*, which we use to generate the never-claims, has even finished! This result is very encouraging, as large formulae with a lot of strong fairness constraints can take quite a long time to convert to a Büchi automaton with *ltl2ba* (SPIN's own algorithm ceases to work on much smaller formulae), while the linear weak alternating automata-based approach can start checking right away.

3.3 An easier label calculation criterion

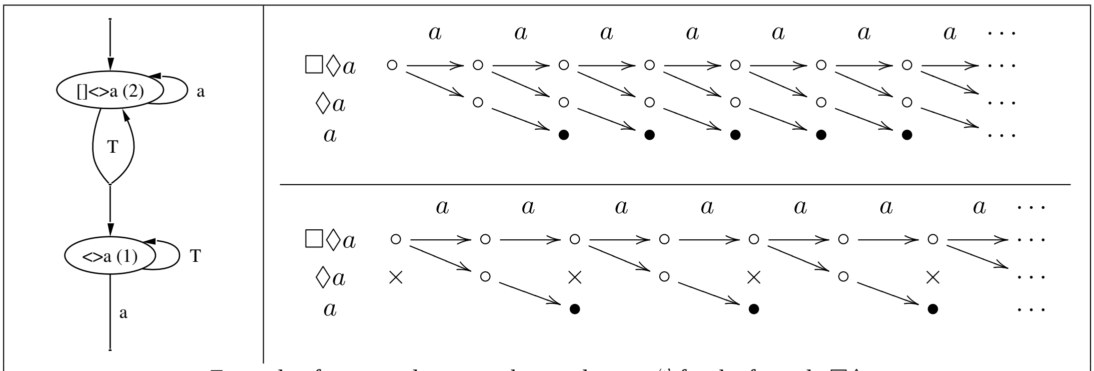
Recall the acceptance criteria of a linear weak alternating automaton: There is a run dag such that every infinite path of this run dag has an even-ranked smallest (and due to the linearity infinitely repeated) location. This may hold true for a run dag that has its odd-ranked locations set in every step - it is still accepting as long as this locations do not self-loop infinitely often. Any infinite path will be forced to leave the odd-ranked location then, and due to the linearity we can conclude the general acceptance from this fact.

Therefore, it does not suffice to check if for a given interval, every odd-ranked location is not set once - it might be set every step and still be the looping interval of a satisfying dag. However, it is desirable to check just for these missing locations: due to using the BDDs to calculate the configuration graph successor nodes, we do not examine the edges of the configuration graph anymore, thus we are unable to distinguish between self-loops and descending edges. We could still use the formula provided by Merz/Sezgin:

$$q \in \lambda(c,c') \quad \text{iff} \quad q \in c \text{ and } \models \left(\bigwedge_{p\in c} \delta^+(p)\right) \wedge \left(\bigwedge_{p'\in c'} p'\right) \Rightarrow l_q$$

But this results in a rather lot of additional BDD operations which we are trying to avoid.

However, as it turns out, any linear weak alternating automaton $\mathcal{A}_\varphi$ for a formula φ without subformulae $\circ\psi$ can be checked with the "missing odd-ranked location" criteria as well. This also holds if every location representing the a subformula $\circ\psi$ is even-ranked. As $\circ$ is commutative with all temporal and nontemporal operators, we can put it right in front of the nontemporal subformulae, where it is guaranteed to be of rank 0. The reason why this works is because the largest fixpoint operators copy the transitions of their subformulae, thus allowing to perform a self-loop instead of forking to any lower odd-ranked state.



Example of two run dags over the word $w = a^\omega$ for the formula $\Box\Diamond a$,
with × indicating a missing odd-ranked location

Actually, this observation relies on the linear weak alternating automaton to be exactly as created by the algorithm given above. Any attempts to minimize the LTL formula have to ensure that all $\circ\varphi$-subformulae are even-ranked, and all attempts to minimize the linear weak alternating automaton must make sure they do not destroy the transitions required.

3.4 Results

Our implementation cannot compete with SPIN in every aspect. It has a much more simplified input language, does not perform partial order reduction or any other kind of simplification. Like SPIN, it performs undirected search, i.e. does not use a heuristics function to order the successors of some state in order to find shorter trails. However, the main difference remains in the amount of memory used. SPIN is much more efficient, and provides both lossless and lossy compression.

Thus, running benchmarks against SPIN results in two problems: To obtain comparable results, we first need to make sure that SPIN does not use techniques that have not been implemented in our version (like the partial order reduction), and second, we need to write the Promela input in a way that makes it comparable to our own implementation.

Our implementation uses actions (with guard conditions) and single bits to represent the transitions of the system. It does not support for higher-level constructs like arithmetics or processes. However, these things can be hand-coded into our system.

A first example is a counter (or multiple, interacting counter) that allowes for easy debugging. For the SPIN version we do not use SPINs own arithmetics (which are translated to use the CPU's arithmetics) but implement the same bit-switching as we do with our own implementation.

We give some pretty unsorted results first:

Model	Model Size	Formula Size	satisfiable	ltl2ba time	SPIN pan time	our time
twocounter	11	37	no	$0.025s$	$0.653s$	$3.621s$
3.2.SAT	21	45	yes	$0.257s$	$7.240s$	$32.542s$
3.3.SAT	21	65	yes	$75.592s$	$22.300s$	$43.908s$
2.2.UNSAT	14	45	no	$0.631s$	$3.960s$	$16.566s$

We can draw some conclusions from this early data: Our implementation is not as fast as SPIN, however with larger search times these disadvantage seems to cease (due to memory constraints, we cannot check larger models yet). The 3.3.SAT-example is very promising: Our implementation finishes searching faster than *ltl2ba* finishes generating the never claim. This raises hope that for large LTL-formulae (we have to admit that, compared to the model, these formulae are really large, as they incorporate a lot of strong fairness constraints) our implementation can start right away, and has the chance of finding a counterexample without having to create the Büchi automaton first.

We try to reproduce that result on a different, more practical model. However, most models available are pushing the limits with respect to the model size, but either do not have "real" temporal properties at all (only checking of safety properties, where a single depth-first search is sufficient) or just pretty small ones that do not incorporate fairness constraints.

We thus modify the dining philosophers model to be checked with large formulae. We claim that if every philosopher attempts to eat infinitely often, philosopher one will (eventually) eat infinitely often. Of course, the standard example with every philosopher starting with his left fork will result in a deadlock, providing a counterexample. To obtain a correct model, the last philosopher is modified to begin with his right fork. We compare a different number of philosophers, both normal and inverted:

Model	Model Size	Formula Size	satisfiable	ltl2ba time	SPIN pan time	our time
dinphil4	16	39	yes	$0.008s$	$0.067s$	$0.125s$
dinphil8	32	71	yes	$34.586s$	$0.142s$	$0.419s$
dinphil9	36	79	yes	$284.170s$	$0.376s$	$0.541s$
dinphil10	40	87	yes	$3686.357s$	$2.023s$	$2.221s$
dinphil12	48	103	yes	DNF[2]		$16.000s$
dinphil15	60	127	yes			$328.081s$
dinphil16	64	135	yes			DNF[1]
dinphil4i	16	39	no	$0.007s$	$0.070s$	$0.128s$
dinphil8i	32	71	no	$35.949s$	$5.218s$	$82.238s$
dinphil9i	36	79	no	$284.172s$	$52.802s$	DNF[1]
dinphil10i	40	87	no	$3686.357s$	DNF[1]	

[1] - out of memory (1 GB primary memory, OS started swapping)

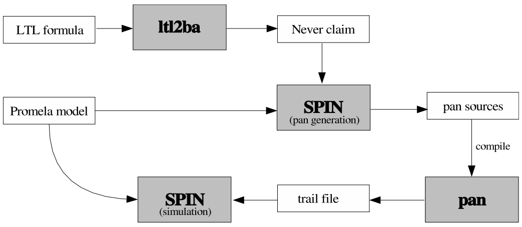
[2] - timeout (would take about a week)

Of course, the generated never claim can be preserved for further tests, which is not possible with our model. However, this example illustrates even more clearly that once the LTL formulae get too big, our implementation is far more efficient on finding counterexamples. It is slower on unsatisfiable models, but memory constraints (mostly imposed by the BDD package) are a bigger problem.
Thus, the next step in evaluating the use of linear weak alternating automata in model checking is an attempt to reduce the memory requirement. Apart from the BDD package, much memory is lost because we cannot use fixed-sized arrays in our model checker, as the size of the automaton is not known at compile time. SPIN evades this problem by generating C code for a model, which can take advantage of knowing the required array sizes at compile time. As a final test of the adaptability of our algorithm, we chose to modify SPIN to use our new algorithm.

4 Adapting SPIN

4.1 SPIN architecture

The SPIN core architecture has been documented in [Hol91], chapter 13. SPIN itself is not a model checker, but a model checker generator. The process of LTL model checking is diverted into some stages: First, the LTL formula is translated into a so-called never claim. This can be done by SPIN itself, but it is highly advisable to use *ltl2ba* instead, as it is much faster on large formulae. Then, the model (written in Promela, short for *Pro*tocoll *me*ta *la*nguage) and the never claim (in fact also written in Promela) are combined, and five source files (named `pan.b`, `pan.c`, `pan.f`, `pan.h` and `pan.t`) are generated. After compiling them with a common C-compiler, the resulting pan (short for *p*rotocol *an*alyzer) executable can be invoked to perform the actual model checking. If a witness trace of a property violation is found, it can than be fed back to SPIN as a “guide” for simulation, which shows the behaviour of the model on its way to the error.



The process of LTL model checking with SPIN

SPIN combines a lot of algorithms, some of them mutual exclusive (like breadth-depth-first search and liveness property checking). Luckily for us, almost all algorithms are generated into the pan sources, where they are enabled or disabled by preprocessor commands; thus all pan sources contain the code for all the algorithms. In other words, the code generated specially for a problem is kept to a minimum.
Promela describes processes that are allowed to run concurrently, access global variables or communicate by using channels. The “class” of a process is called the “proctype”. For any given proctype, a number of processes can be instantiated.
The never claim itself is nothing but a special proctype, which has only one instance. Instead of doing synchronized automaton- and model-steps, SPIN takes the steps alternatingly. This also allows SPIN to handle atomic sequences easily: While inside an atomic block the claim-steps are skipped. Although each step in the atomic block is performed sequentially, it appears as one atomic step to the automaton. Likewise, channel rendevouz blocking is done just like that: the initiating process moves, and after that all movement (including the never claim) not contributing to the rendevouz is suspended.
This architecture is both positive and negative for our attempt to combine our approach with SPIN: As the automaton is handled as a “slightly special” process, we can easily find the places where it is accessed and do not have to change a lot of code. As a matter of fact, there is little need to skip that “slightly special process” approach at all: We can adopt it and just change the depth-first search where the label criterium has to be added.
The disadvantage of using a process for the automaton is that this does not work with alternating automata. A process as well as a Büchi automaton have one single state, and for every state they do have a number

of transitions, which lead to other states. An alternating automaton, on the other hand, has a set of active states, and the transitions lead to other sets. By identifying a set of states with a distinct state, we build our on-the-fly generalized Büchi automaton; we cannot use a (precompiled) process for that. Instead, the set of successor sets has to be calculated dynamically (and this is what our whole approach is all about). Thus, our automaton process has to be a little more special than the never claim process was. Actually, it has to calculate the transitions (normally precalculated by SPIN and stored in `pan.t`) on-the-fly.

4.2 Inserting our algorithm

We insert the code in two successive steps. First, we replace the CVWY-algorithm with a special kind of Tarjan's algorithm. This requires a few changes:

- We have to add a second stack, storing the SCC member candidates.
- We have to add a reference from the hash table entries to that second stack to recognize members of the currently evaluated SCC, as well as closed SCCs (it is possible to find edges leading into an already closed SCC).
- We have to add two blocks of code; one that predates the root candidate and checks the (right now, only one) acceptance condition, right after the recursion was conducted, and another one right before stepping out of the recursion, which processes completely explored SCCs.

This changes are all to be made to static code and do not require model-specific changes.

The second step replaces the use of a never claim with the on-the-fly calculation of configuration graph nodes. This requires a fair amount of precalculated code, which is concerned with calculating the successor configurations, extracting the label and evaluating them and many other things. We chose to abandon the transition table for the linear weak alternating automaton moves, and use a lot of case differentation to handle the normal processes and the automaton. This step also requires the inclusion of a BDD package, so we wrote a small BDD package specialized for our needs, which also helps to reduce memory requirements.

4.3 Results

4.3.1 SPIN and the Tarjan algorithm

In the first stage of the adaption, we enhanced SPIN with the Tarjan algorithm. Like discussed above, there are no dynamic changes (generated code dependent on the model used) and little static changes to be made, and the algorithm fits quite well into the concept of SPIN.

We use some models to compare both algorithms:

Model		CVWY algorithm			Tarjans algorithm		
name	SAT?	time	states	max depth	time	states	max depth
dinphil10	yes	0.50	129058	258094	0.57	129058	258094
dinphil8i	no	7.30	774413	320719	7.37	774413	320719
mobile1	no	0.57	44455	1833	0.53	42807	1667
mobile2	no	0.15	16191	1942	0.16	15932	1874
Steam Generator Control [Zha99]	yes	0.16	3115	11478	0.21	3115	11478
Needham-Schroeder-Protocol [MS02]							
Model 1, fixed	no	0.10	379	30	0.11	379	30
Model 2, fixed	no	0.60	7179	43	0.49	7179	43
Model 2, original, Claim 2	yes	0.8	2593	43	0.8	2591	43

These values are not very spectacular - most of the time, we see the same depth and state number - as expected, as the complete state space needs to be explored to verify correctness. When detecting errors, the algorithm takes the same path, but instead of searching for cycles, it searches for SCCs. Therefore, it should find the same witness cycle anyway. After all, both algorithms do not seem to be very different at all.

This contradicts the results of Geltenhuys/Valmari [GV04], which reported their algorithm to be much faster than the CVWY algorithm, although they do not mention which implementation they used to obtain the CVWY times, and they admit their approach is not likely to be very practical, as it did not involve on-the-fly statespace exloration.

4.3.2 SPIN with linear weak alternating automata

Currently, the linear weak alternating automata extension to SPIN is under testing, and there are still some issues to be resolved: Currently, neither memory nor time are a problem, but the C code we generate gets too big to be compiled.
First tests indicate that the SPIN-linear weak alternating automata combination is faster than our own approach, but is not as fast as the original SPIN algorithm. Until the C code becomes uncompileable, we also use much less memory. If these results hold and we find a way to generate less code, we can hope to verify the results we had with our own implementation and be able to check larger models.
We will present our results in our final thesis, however we can expect them to be similar to the ones we already obtained.

5 Discussion and further work

At this point of writing, we can assume that out algorithm does not provide an improvement to SPIN for average case problems. However, for large LTL formulae, significantly lower search times can be archived. Currently, most examples of Promela models either do only use safety assertions, or they use very small LTL formulae. Our approach might be encouraging to use bigger LTL formulae, which could be useful for abstraction. As to our knowledge, little research has been done about this abstraction possibility.
For the remaining time of this thesis, we will attempt to solve two problems with our algorithm in SPIN:

- Extracting counterexamples is far more difficult, and has not yet been implemented. Because of using multiple acceptance criteria, the recursion stack does not neccesarily contain a complete loop of the SCC. The SCC-stack, however does not contain a continuous path of the product automaton. Therefore, a more sophisticated method of extracting the counterexample has to be devised.

- The BDD package imposes a big problem due to the unpredictable amount of memory required and time consumed. For very large models, the BDD package slows the checking process down to a point of infeasibility. Therefore it seems profitable to abandon the BDD package in favour of a more direct approach.
  The BDD package is used to combine all the possible successor state sets for a given set of linear weak alternating automaton locations. It helps us by identifying minimal successor sets. However, we can calculate all combinations directly and identify non-minimal successors by finding subsets within the list of other successors. We will attempt to find an algorithm that is faster or at least more predictable than the BDD package we use right now.

Of course, there are many topics beyond the scope of this thesis, some of which include:

- Bit-state hashing is compromised by our algorithm because of the necessity of storing a reference to the SCC-stack element in the hash table. We might, however, use a second hash-table as an access index on the SCC-stack, and use bit-state hashing on the main hashtable. This approach relies on the assumption that SCCs are usually much smaller than the complete state set. In our experiments, we found this to be true for some models, whereas other models in fact provided a SCC containing the whole state space. Whether bit-state hashing is feasible in such an environment, and which models do provide us with a number of smaller SCCs instead of a single big one, remains to be seen.

- Heuristic search is traditionally concerned with selecting the best model action. However, using generalized Büchi automata provides us with some additional information that gets discarded in the translation of generalized Büchi automata to nondeterministic Büchi automata: the multiple acceptance conditions. Although our naive attempt to use this additional information (by selecting the claim move that leads towards a not yet satisfied acceptance condition) failed, there might still be some potential to use that information.

- Minimization of the linear weak alternating automaton: Our attempts at this indicate this is hard and most likely infeasible to do. There are a lot of approaches towards LTL formula minimization (see [SB00] for an interesting approach we implemented in our own model checker), but little has been done on linear weak alternating automata that claims to be very successful. However, there are good minimization algorithms for generalized Büchi automata, including delayed simulation [Fri03]. They might be applied to the on-the-fly automata generation as well.

References

[BCRZ99] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1633:60–71, 1999.

[CKSO04] Edmund M. Clarke, Daniel Kroening, Ofer Strichman, and Joel Ouaknine. Completeness and complexity of bounded model checking. In *5th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004. to appear.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.

[CVWY92] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[Fri03] Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In Oscar H. Ibarra and Zhe Dang, editors, *Implementation and Application of Automata. Eighth International Conference (CIAA 2003)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35–48, Santa Barbara, CA, USA, 2003.

[FS01] Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. In *Proceedings of the First International Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[FW03] Carsten Fritz and Thomas Wilke. Simulation relations for alternating Büchi automata. *Theoretical computer science*, November 2003.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[GV04] Jaco Geldenhuys and Antti Valmari. Tarjan's algorithm makes LTL verification more efficient. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer-Verlag, March-April 2004.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[Hol03] Gerard J. Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley, 2003.

[May98] Richard Mayr. Strict lower bounds for model checking BPA. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 18, 1998.

[McM92] Kenneth L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[Mer00] Stephan Merz. Weak alternating automata in Isabelle/HOL. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2000.

[MS84] David E. Muller and Paul E. Schupp. Alternating automata on infinite objects: Determinancy and rabin's theorem. In *Proceedings of the Ecole de Printemps d'Informatique Theoretique on Automata on Infinite Words*, volume 192 of *LCNS*, pages 100–107. Springer, May 1984.

[MS87] David E. Muller and Paul E. Schupp. Alternating automata on infinite trees. *Theoretical computer science*, 54(2/3):267–276, October 1987.

[MS02] Paolo Maggi and Riccardo Sisto. Using spin to verify security properties of cryptographic protocols. In *9th International SPIN Workshop on Software Model Checking*, pages 187–204, 2002.

[MS03] Stephan Merz and Ali Sezgin. Emptiness of linear weak alternating automata. http://www.loria.fr/ merz/papers/waa-emptiness-report.pdf, 2003.

[NSS94] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[Pnu77] Amir Pnueli. The temporal logic of programs. *In Proceedings of the 18th IEEE Symposium Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from ltl formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 247–263. Springer Verlag, 2000.

[SC85] A. Prasad Sistla and Edmund M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. In *SIAM Journal of Computing*, pages 146–160. SIAM, 1 edition, 1972.

[Tau03a] Heikki Tauriainen. Nested emptiness search for generalized Büchi automata. Research Report A79, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, July 2003.

[Tau03b] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7, pages 389–455. Springer, 1997.

[Var95] Moshe Y. Vardi. Alternating automata and program verification. In *Computer Science Today*, pages 471–485. Springer, 1995.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[Zha99] Wenhui Zhang. Model checking operator procedures. In *International SPIN Workshop*, 1999.