Model Checking with the Sweep-Line Method

Master's Thesis in Software Engineering

Andreas Lilleskare
Western Norway University of Applied Sciences
&
University of Bergen

Supervisors
Lars Michael Kristensen & Sven-Olai Høyland
Western Norway University of Applied Sciences

June 2017

Abstract

Explicit-state model checking is a formal software verification technique that differs from peer review and unit testing, in that model checking does an exhaustive state space search. With model checking one takes a system model, traverse all reachable states, and check theses according to formal stated properties over the variables in the model. The properties can be expressed with linear temporal logic or computation tree logic, and can for example be that the value of some variable x should always be positive. When conducting an explicit state space exploration one is guaranteed that the complete state space is checked according to the given property. This is not the case in for instance unit testing, where only fragments of a system are tested. In the case that a property is violated, the model checking algorithm should present an error trace. The error trace represents an execution path of the model, demonstrating why it does not satisfy the property. The main disadvantage of model checking, is that the number of reachable states may grow exponentially in the number of variables. This is known as the state explosion problem.

This thesis focuses on explicit-state model checking using the sweep-line method. To combat the state explosion problem, the sweep-line method exploits the notion of progress that a system makes, and is able to delete states from memory on-the-fly during the verification process. The notion of progress is captured by progress measures. Since the standard model checking algorithms rely upon having the whole state space in memory, they are not directly compatible with the sweep-line method.

We survey differences of standard model checking algorithms and the sweep-line method, and present previous research on verifying properties and providing error traces with the sweep-line method.

The new contributions of this thesis are as follows: (1) We develop a new general technique for providing an error trace for linear temporal logic properties, verified using the sweep-line method; (2) A new algorithm for verifying two key computation tree logic properties, on models limited to monotonic progress measures; (3) A unified library for the sweep-line method is implemented with the algorithms developed in this thesis, and the previous developed algorithms for verifying safety properties and linear temporal logic property checking. All algorithms implemented, are validated by checking properties on a model of a stop-and-wait communication protocol.

Acknowledgments

Foremost, I would like to thank my supervisors Lars Michael Kristensen and Sven-Olai Høyland for valuable input regarding problems, and a thorough follow up in the writing of this thesis.

I would like to thank Linn-Kristine Glesnes Ødegaard for proofreading, and my fellow master's students, especially Anders Kvalvaag, Anders Flemmen and Marius Kalvø for motivation and feedback.

Contents

Chapter 1

Introduction

Program malfunction can be both dangerous and costly. On example is the radiation therapy machine Therac-25 [1] in which patients were given a massive overdose of radiation. Another example is the bug in the Intel Pentium floating-point division unit [2] that cost Intel a financial loss of about 475 million US dollars to replace the faulty processors.

As stated in [3] by Baier and Katoen, the major software verification techniques used in practice today are peer review and software testing. They observe that while these techniques combined can detect many flaws, subtle errors such as concurrency and communication defects are hard to detect. Furthermore, testing can only show the presence of errors, not prove their absence.

This master's thesis focus on formal verification techniques based on the principle of *explicit-state model checking* with key interest in the sweep-line state space exploration method [4].

1.1 Explicit-State Model Checking

Formal verification techniques aim to prove software correctness with mathematical rigour. Explicit-state model checking is one of these techniques. It is based upon the core idea of conducting an explicit state space exploration of a system model, and thereby traverse all reachable states in the model. Figure 1.1 shows a simple overview of the explicit-state model checking process. It depicts that one starts with a system model, from which a *transition system* is obtained by *unfolding* all choices of enabled actions in each state. The unfolding captures all reachable combinations of variables, and transitions that is in the model.

The model checking algorithm explores the transition system. In the exploration,

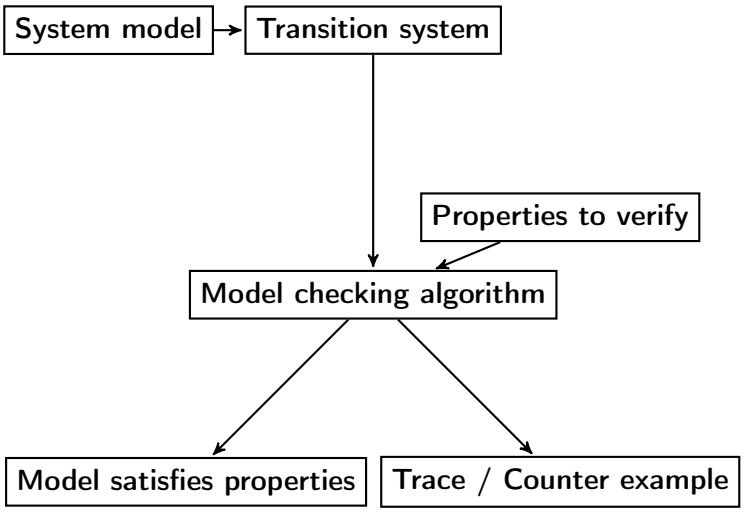


Figure 1.1: A schematic overview of explicit-state model checking.

all states are checked against formally stated properties, expressed using for instance propositional or temporal logic. The model under assessment should be a correct and unambiguous model representing a piece of software, protocol or algorithm. Informally, the transition system is a directed graph, where nodes represent the system states, and edges represent the state transitions that make it possible for the system to change its state. The values of the variables, in addition to the program counter makes up each system state, and it is in terms of these variables one expresses properties. Propositional logic is concerned with truth values, i.e., whether some proposition is true or false in a given state. Temporal logic is concerned with propositions that are qualified in terms of time, e.g., a proposition is *always* true or a proposition will *eventually* be true.

Algorithm 1 Basic state space traversal algorithm

```
 1: visited                                   ▷ Set of visited states
 2: E                                         ▷ Set of edges in the state space
 3: U                                         ▷ Data structure for unprocessed states
 4: init                                      ▷ Initial state to start the traversal from

 5: procedure TRAVERSE(init)
 6:     visited ← init
 7:     U.enqueue(init)
 8:     while U ≠ ∅ do
 9:         s ← U.dequeue()
10:         for all (s, s') ∈ E do
11:             if s' ∉ visited then
12:                 U.enqueue(s')
13:                 visited ← visited ∪ s'
14: end procedure
```

The basic algorithm for a state space exploration is presented in algorithm 1. It starts with an initial state (init), and inserts this to a data structure representing unprocessed states (line 7). Each state is then processed by dequeuing it, and adding all successors that have not been visited, to the queue of unprocessed states (line 9 to 12). By continuing this way until the queue of unprocessed states is empty, one is guaranteed to process all states that are reachable from the initial state. Since each state is marked when visited (line 13), they are only processed once, leading to a time complexity that is linear in the number of reachable states from the initial state. Depending on the data structure that is used to implement the unprocessed states ($\mathcal{U}$), different state space traversals can be obtained, i.e., a stack would produce a depth-first traversal whereas a queue would produce a breadth-first traversal.

Spin [5] and Lola [6] are two examples of software tools for model verification. These tools provide different state space traversal algorithms, and performs verification of properties. Examples of interesting properties that can be verified are:

- The value of the variable x is always positive.
- At most one process can be in the critical section at any time.
- For every state there exists an enabled transition (i.e., no deadlocks).
- Each process will enter the critical section infinitely often (i.e., liveliness).

An important feature of model checking is that in the case a property is violated, then one can obtain an *error trace* (counter example). This trace demonstrates why the property does not hold, and can for instance lead to an undesired state. Error traces can be used to understand and correct errors in the software design.

The main disadvantage of explicit-state model checking is the state explosion problem [7]. In short, the state explosion problem is that the number of reachable states in a model can become infeasible to handle as it may grow exponentially in the number of variables or processes in the model. To combat the state explosion problem, several techniques have been developed. One of these is the *sweep-line method* [4].

1.2 The Sweep-Line Method

The sweep-line method is a state space traversal technique that exploits a notion of progress that a system may make. This is done by assigning a *progress value* to all states which quantifies the amount of progress the system have made in each state. The state space is then divided into layers, where all states with the

same progress value are in the same layer. The sweep-line method limits the state space exploration to one layer at a time. When all states in the current progress layer have been explored, they are removed from memory. This reduces the peak memory usage as only a fragment of the states are stored in the memory at any given time. The sweep-line method traverses the state space in a *least-progress-first order*. To terminate, previously visited states must be recognized. The sweep-line method does this by marking all newly encountered states with less progress (meaning that there is a state that has a successor state with strictly less progress) as *persistent*. Persistent states cannot be deleted from memory, and thus it avoids an infinite loop in the exploration of the state space. States can be visited several times in the presence of persistent states. The sweep-line method therefore uses more time, in favour of being able to reduce peak memory use, compared to standard state space exploration.

Because the sweep-line method deletes the states after processing them, it has some limitations compared to the standard model checking algorithms [3, p. 161, p. 202, p. 336] that rely on *depth-first search* (DFS) or *breadth-first search* (BFS). There have been several papers investigating extensions to the sweep-line method, and some of them are summed up in [4]. The sweep-line method must use tailored algorithms to check properties, and until now there has been no coherent presentation of developed algorithms for model checking with the sweep-line method.

1.3 Models and State Spaces

When verifying a program with explicit-state model checking, one needs a model representing a program. Such a model specifies which operations the program exhibits, and what the resulting state is after each operation, i.e., which variables are affected. A model is a high-level description of how a program is designed. Promela [8] and Petri nets [9] are modelling languages used to specify models for the Spin and Lola tools, respectively.

When representing a concurrent system, *transition systems* are often used [3, p. 19]. A transition system is a directed graph where the nodes represent the states, and edges model the valid transitions from one state to another. A state represents properties of a system at a certain moment of the system's execution.

Figure 1.2 shows a model of a simple program that starts in a waiting state, shown with a horizontal start arrow, then enters and leaves a critical section, before it is done. The labels on the edges are the *actions* that the program performs, and the labels in the nodes are the *states* of the program. The transition system starts in an initial state and can move from one state to another according to a

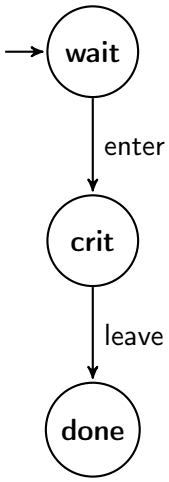


Figure 1.2: Example of a simple model that enters and leaves a critical section.

transition relation. If the transition system has more than one initial state, or more than one successor with the same action, a non-deterministic choice is made. A set of atomic propositions are defined in regards to which properties the system possesses, and a labelling function marks each state with the atomic propositions that holds in each state.

Figure 1.3 shows the resulting transition system of a *parallel composition* of two processes of the simple model shown in figure 1.2. The resulting transition system is a subset of the Cartesian product of the two processes, called the *product transition system*. The labels in the nodes of the transition system are the combined state of the two processes. For example, the state labelled (wait1, wait2) represents that both process 1 and process 2 are in their wait state, and likewise the state labelled (crit1, done2) represents that process 1 is in the state crit, and process 2 is in the state done. The labels on the edges represent the action performed, i.e., (1 enter) represents that process 1 enter the critical section whereas (2 leave) represents that process 2 leaves the critical section.

An execution of a transition system results from the resolution of the non-determinism in the system. This way a transition system describes the possible behaviours of the system [3, p. 24]. By considering either the actions or atomic propositions made in a run of the transition system will generate a word when executed. All possible runs a transition system can make, represents the *language* of that system.

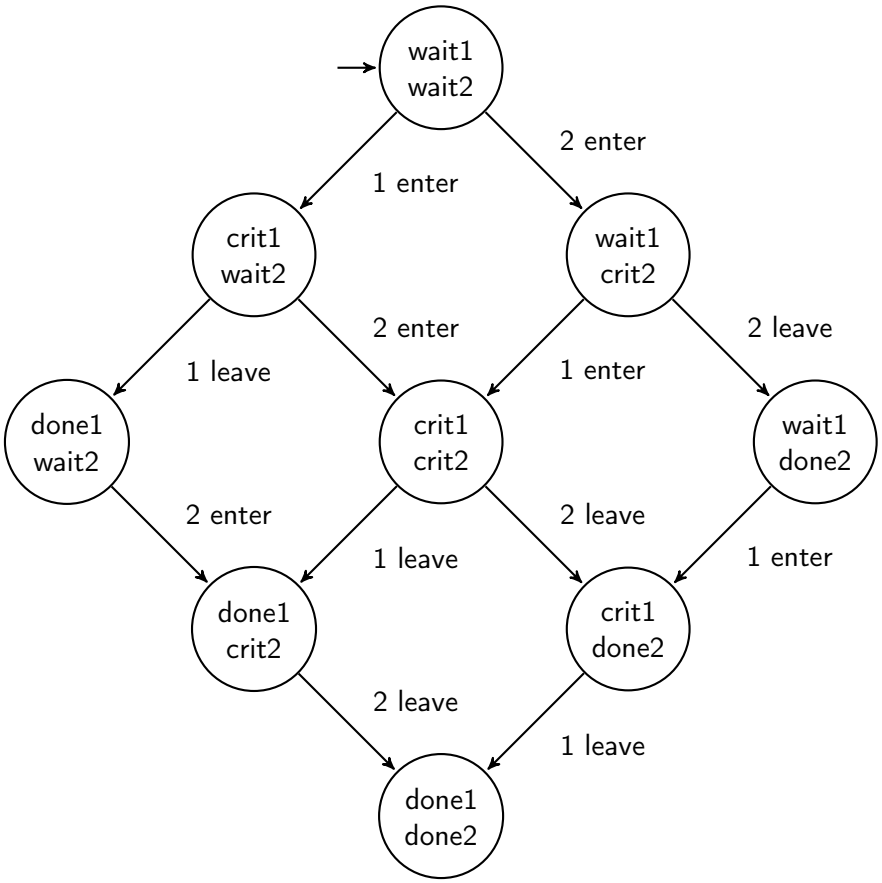


Figure 1.3: Resulting transition system after parallel composition of two processes of the model in figure 1.2.

1.4 Automata and Linear Temporal Logic

Automata are used to recognize the languages of transition systems. An automaton has a set of states, with at least one initial state, and a set of acceptance states. It can move from one state to another according to a transition relation. An automaton reads a word w and either accepts or rejects it. The automaton starts in an initial state and reads one letter at a time from w. For each letter processed, it moves according to the transition relation until the word is read. This is called a run for the automaton. A word is *accepted* if the automaton can read the whole word, and end in an acceptance state. If the automaton does not have a legal transition for the letter that should be processed, or if the automaton does not end in an acceptance state, then the word is *rejected*. Figure 1.4 depicts an automaton that accepts all words of the regular language of: a*bb*, which represents zero or more a's followed by at least one b. The initial state of figure 1.4 is marked with a start arrow, and the acceptance state is indicated using a double circle.

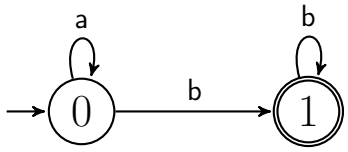


Figure 1.4: A schematic of an automaton accepting the regular language of: a*bb*.

Regular languages can be recognized by non-deteministic finite automaton (NFA) and are used in explicit-state model checking to verify properties whose error traces constitute a regular language [3, p.149]. NFA accepts words of finite length. To verify a broader spectrum of properties one must use *non-deterministic Büchi automata* (NBA) which accepts languages of infinite words [3, p.167]. An accepting run in an NBA is a run where the NBA is in an acceptance state infinitely often.

Linear temporal logic (LTL) is used to specify properties, and an NBA can algorithmically be constructed from formulas written in LTL [3, p. 267]. LTL properties are specified over the atomic propositions in a system. The basic operators in LTL are:

$\Diamond\ \Phi$, reads *eventually* (Φ holds now or eventually in the future)
$\Box\ \Phi$, reads *always* (Φ holds now and forever in the future)
$\bigcirc\ \Phi$, reads *next* (Φ holds in the next state)
Φ_1 U Φ_2, reads *until* (Φ_2 holds now, or Φ_1 holds until Φ_2 holds)

These operators can be combined with each other, and logical connectives (e.g., $\vee$, $\wedge$, $\neg$) such that one can express different properties about the paths (words) of a system. In figure 1.5, based on [3, p. 229], some properties that can be expressed using LTL are visualized

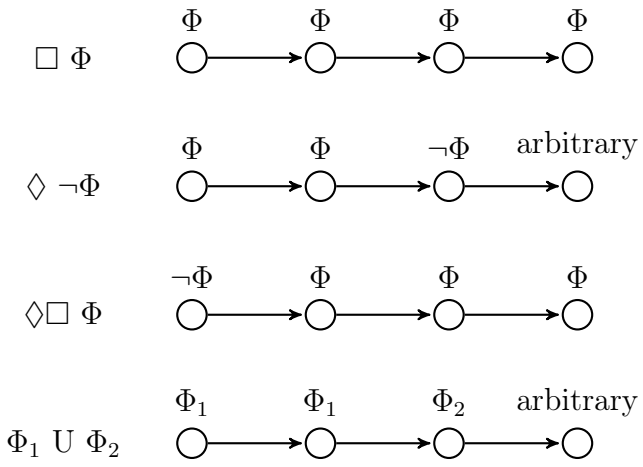


Figure 1.5: Example of LTL properties.

From top to bottom in figure 1.5, there are paths satisfying the properties:

Always Φ: Φ holds in every state
Eventually not Φ: Φ does not hold in one state along the path
Eventually always Φ: at one point Φ holds forever
Φ_1 *until* Φ_2: Φ_1 holds in all states until Φ_2 holds.

1.5 Computation Tree Logic

Computation tree logic (CTL) properties differs from linear temporal logic (LTL) properties in that LTL is linear and path based, whereas in CTL the notion of time is based on a branching of successor states. This results in a *tree* describing possible executions. Each sub tree rooted in a state s represents all computations possible from that state [3, p. 311]. CTL therefore enables us to state properties of *some* paths of the system under evaluation. Some properties can be expressed with both LTL and CTL, like regular safety properties. However, some properties can only be expressed using LTL and not in CTL and vice versa [3, p. 330]. An example of a property that one can express using CTL, but cannot be expressed in LTL is "for every computation it is always possible to return to the initial state" [3, p. 310].

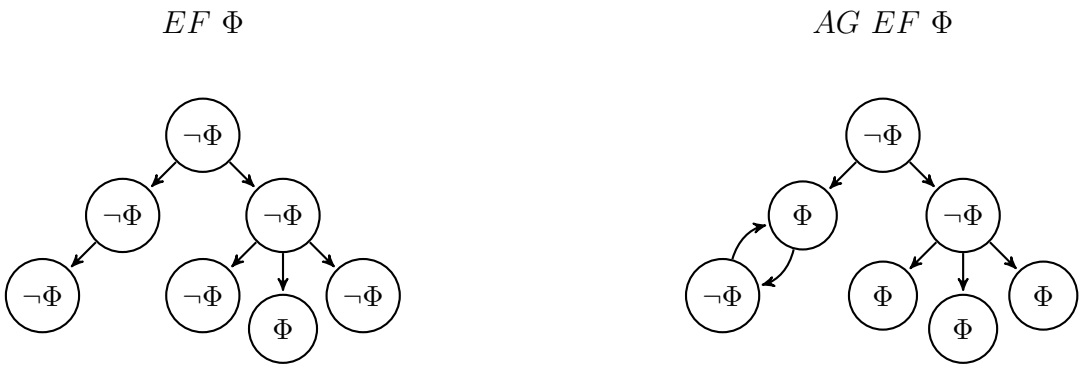
CTL expresses properties over a branching notion of time, and uses two path quantifiers to express properties rooted at a state s:

A Φ, reads *all paths* (Φ must hold in all computations possible from s)
E Φ, reads *exists path* (Φ must hold in at least one computation possible from s)

In addition to the operators used in LTL:

G Φ, reads *globally* (equal to $\Box$)
F Φ, reads *finally* (equal to $\Diamond$)
X Φ, reads *next* (equal to $\bigcirc$)

These operators can be combined with each other, the until operator, and the logical operators as with LTL. Figure 1.6 depicts two CTL properties. Figure 1.6a illustrates the property that "there exists a computation where φ eventually holds". Figure 1.6b illustrates the property that "for all computations one eventually reach a state where φ holds".



(a) Exists finally Φ

(b) All global exists finally Φ

Figure 1.6: Examples of CTL properties.

1.6 Research Method

A focus of this thesis is the model checking algorithms compatible with the sweep-line method. We want to investigate its algorithms, and gain knowledge of efficient data structures and implementation aspects. We rely upon both a formal algorithm design methodology, but also the experimental methodology of algorithm engineering.

In the first chapter of [10], a methodology for algorithm research and analysis is introduced. In the design of algorithms, one is abstracting implementation details and prove correctness mathematically. The running time of an algorithm is expressed in asymptotic worst case scenarios called *big-Oh* notation. In big-Oh notation one is concerned about the time and space complexity of an algorithm in terms of the size of the input. Using big-Oh notation one has abstracted from specific hardware details by expressing the complexity over primitive operations, e.g., assignments, comparisons and calling methods. The authors of [10] argues that experimental evaluations have some limitations. Experimental tests can only be done on a limited input set, and to compare the running time of two algorithms, they must be tested on exactly the same hardware and software environments. Formal algorithm design has the advantage that it accounts for all possible inputs when expressing time complexity and is independent from the environment an algorithm is executed. When developing an algorithm one must define the *problem* that the algorithm should solve, and specify the input and output requirements. One uses pseudo-code to capture the step-by-step solution to a problem.

When implementing code for a specific problem, one is interested in the performance of the code in a practical manner. This could mean that potentially even an algorithm, with higher worst-case asymptotic performance is preferred over others if it in practice performs better. This means that the worst-cases either are

not present in the current context or occurs with significant low frequency so that the total average is tighter to the lower bound of the algorithm. Quick-sort [11] is an example where despite its worst-case asymptotic performance, $O(n^2)$, it can often outperform other sorting algorithms like merge-sort which has a worst-case asymptotic performance of $O(n\ log\ n)$. We therefore also want to use algorithm engineering as a methodology in this thesis, when investigating algorithms for the sweep-line method.

Algorithm engineering can be viewed as general methodology for algorithmic research is a cycle consisting of algorithm design, analysis, implementation and experimental evaluation that resembles Popper's scientific method [12]. Since algorithm engineering is always driven by real world applications, the methodology is a feedback loop of analysis, implementation and evaluation.

A key difference between algorithmic research and algorithm engineering is that research in algorithm theory stems from mathematics and uses deductive reasoning whereas algorithm engineering is driven by falsifiable hypotheses validated by experiments and inductive reasoning [12]. Furthermore, in order to reduce the gap between algorithm theory and practice, realistic computer models with realistic input data must be used in the analysis and evaluation. Figure 1.7 displays the cycle of the algorithm engineering methodology. The conventional asymptotic algorithm work is depicted as the "analysis" part in figure 1.7 which delivers worst-case performance guarantees. In algorithm engineering, one implements the algorithms against a specific application, and conducts experiments with data from that application. Depending on the results one re-implement, or use different data structures to obtain the most desired performance for an application. In [13] they also focuses on implementation of more generic libraries to support re-use implementations that have been proven efficient.

1.7 Research Questions and Results

The main focus of this master's thesis is the sweep-line method which differs from standard model checking techniques in that it deletes states from memory on-the-fly in the verification process. Many model checking techniques rely upon depth-first search (DFS) or breadth-first search (BFS) traversal and are therefore not compatible with the *least-progress-first* exploration defined by the sweep-line method. Several ideas and improvements have been suggested to the sweep-line method to be able to verify the properties and providing error traces, in a similar way as supported by standard verification methods. "The Sweep-Line State Space Exploration" [4] sums up some of the most recent work done on the sweep-line method. While there are different papers investigating and assessing these improvements, a consistent implementation of the sweep-line method and

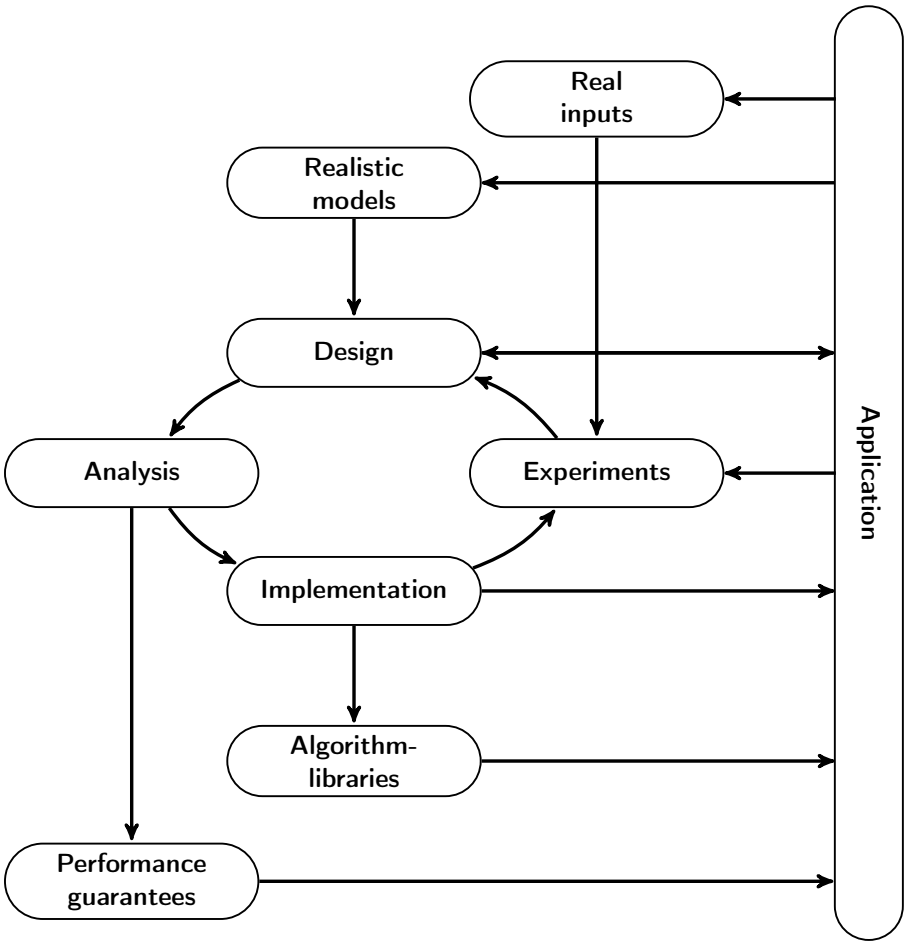


Figure 1.7: Algorithm engineering cycle following [12].

its improvements has not been made. Implementation of the different variations in a unified way, and investigate how to support property checking and providing error traces is a goal of this thesis.

The research questions that arise are:

1. Property checking of the sweep-line method compared to other state space traversal methods such as DFS or BFS
   (a) What is the main difference with the sweep-line method compared to the standard techniques?
   (b) How to support the verification of different sets of properties?
   (c) How to provide error traces?
2. Unified implementation
   (a) How to implement the different variations in a unified way?

(b) Which interfaces must the sweep-line method depend on for the representation of a model, the property to be verified and memory management?

It is important to gain knowledge of the differences between standard techniques and the sweep-line method when developing tailored algorithms for it. The sweep-line method is used for state spaces that one is not able to store in memory under the verification process, but nevertheless one wants to be able to perform the verification of the desired properties and providing error traces when a property is violated. Because of the very large state spaces in many models under verification, highlighting of the interfaces and overhead usage by the sweep-line method can lead to more efficient implementations.

The main contributions of the work conducted in this thesis are:

- A new general algorithm for providing error traces when checking linear temporal logic (LTL) properties with the sweep-line method.
- A new algorithm for checking important computation tree logic (CTL) properties.
- Correction of an error in the pseudo code in [4] that lead to higher peak memory usage and premature termination.
- Definitions of the interfaces that the sweep-line method must rely upon for conducting state space traversals.
- Implementation of the algorithms presented in this thesis, that results in a command line tool for conducting property checking.

A unified library with the sweep-line method and its extensions, is also a product of this master's thesis. The implementation is publicly available at [26].

1.8 Thesis Outline

Below is a brief summation of the individual chapters that constitute this thesis.

Chapter 2 - State Space Exploration and the Sweep-Line Method
Introduces the basic sweep-line method, its limitations and how it is used in explicit-state model checking. The different extensions proposed to improve the sweep-line method are also described.

Chapter 3 - Automata-Based Property Checking
Discusses automata-based property checking algorithms for the sweep-line method.

Chapter 4 - Computation Tree Logic Property Checking

Discusses computation tree logic (CTL) checking algorithms for the sweep-line method.

Chapter 5 - Computation Tree Logic Property Checking with Monotonic Progress Measures

Development of a new algorithm for checking two key CTL properties when considereing monotonic progress measures.

Chapter 6 - Error Traces with the Sweep-Line Method

Discusses how to provide error traces with the sweep-line method. A new general algorithm for providing an error trace with linear temporal logic verification is developed. An algorithm for providing an error trace for the CTL properties introduced in chapter 5 is also developed.

Chapter 7 - Implementation

Elaborates on the architecture of the command line tool developed, and design choices made.

Chapter 8 - Experimental Validation

Presents experimental validations with the command line tool developed in this thesis.

Chapter 9 - Conclusion and Future Work

Summarizes the main findings in this thesis and highlights some of the possible directions for further research on the sweep-line method.

Chapter 2

State Space Exploration and the Sweep-Line Method

The sweep-line method is a state space traversal technique that is used for state space exploration in explicit-state model checking. Stated in [4], the main disadvantage with explicit-state model checking is the *state explosion problem* and the majority of model checking research has been in techniques to combat this problem.

The sweep-line method guarantees to explore the complete state space, and by deleting previously visited states from memory it is able to only have a fragment of the state space in memory at any time.

2.1 The Basic Sweep-Line Method

As previously stated, the sweep-line method differs from other state space traversal algorithms like *depth-first search* (DFS) and *breadth-first search* (BFS) by dividing the state space into layers. The notion of progress that the system under evaluation makes is exploited in order to divide the state space into layers. In a model of for instance a simple send-receive protocol, one way of dividing the state space into layers would be to quantify the progress made in terms of the numbers of packets received so far for each state.

The sweep-line method traverses the state space in a *least-progress-first* manner, i.e., one layer at the time. For the simple state space depicted in figure 2.1 this would mean that state 1 is processed first as it is in the first layer, followed by states 2, 3 and 4 because they are in the second layer and so on. In-layer traversal order is not specified by the sweep-line method and one is free to choose any traversal order. The algorithm optimistically assumes that the system always

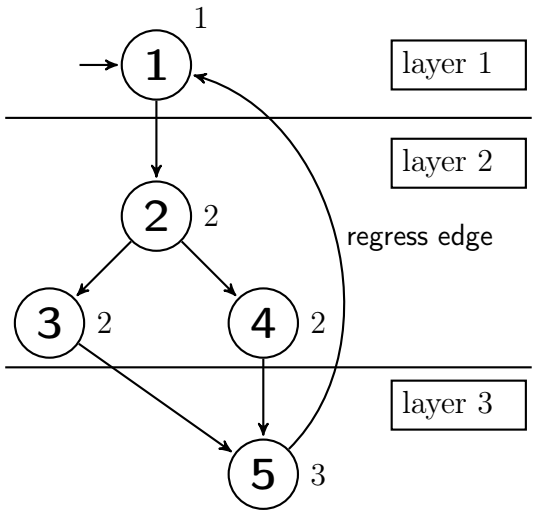


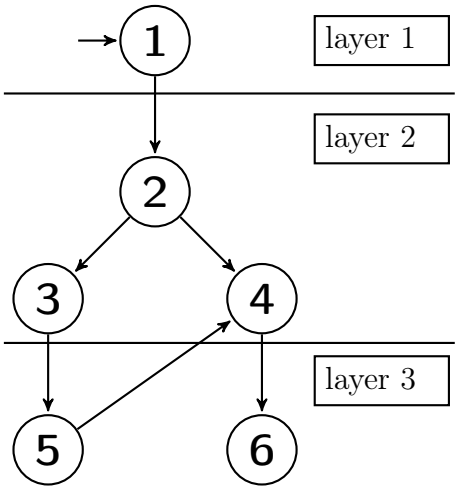
Figure 2.1: An example of a state space divided into three different layers. The progress value of every state is written to the right of each state.

makes progress, and that it can safely delete visited states from memory that belongs to the previous layer, when entering a new one. This is a safe assumption with respect to termination if the system always makes progress which implies that the state space has no *regress edges*, i.e., edges leading from a state with a high progress value to a state with a lower progress value. However, if a regress edge does exist, this can lead to a successor state that has been deleted from memory resulting in an infinite loop in the state space exploration. This is the case in figure 2.1 with the regress edge from state 5 to state 1. The sweep-line method handles this by marking all successors states of regress edges as *persistent*, meaning that it cannot be deleted from memory. Persistent states are recognized as previously visited, if encountered more than once.

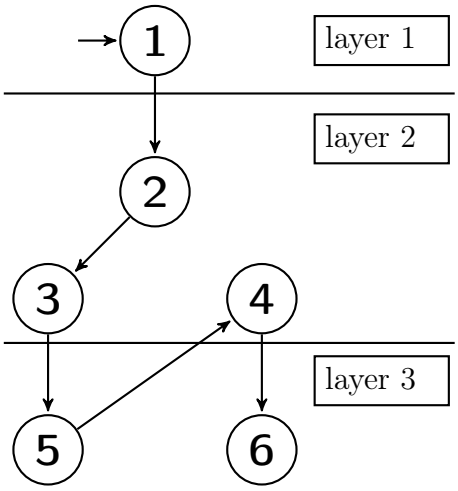
When a regress edge is encountered, there is no way of knowing whether the destination state has already been processed (illustrated in figure 2.2), and one must instantiate a new sweep from that state to make sure that all reachable states are processed. To guarantee that the whole state space is visited, the sweep-line method must re-explore parts of the state space when the state space has *regress edges*. This is done by marking newly discovered persistent states as *roots* for the next sweep.

At any time, only states in one layer plus immediate successors that are in subsequent layers and persistent states are stored in memory. This results in a lower peek memory usage, but in the presence of regress edges; states can be explored several times, which in turn yields longer verification times.

The pseudo-code for the sweep-line method using set notation is presented in algorithm 2. For readability, a set containing just one element is written without



(a) Node 4 is first recognized after state 2 when exploring layer 2.



(b) Node 4 is only recognized after the regress edge from state 5 when exploring layer 3.

Figure 2.2: Example demonstrating that it is impossible to differentiate on which successor states of regress edges are previously visited and which are not when they have been deleted from memory.

enclosing curly brackets, i.e., x instead of {x}. $\psi(s)$ denotes the progress value of state s. The *progress measure* is defined as in [4]:

Definition 2.1 (Progress Measure)
A progress measure is a tuple $\mathcal{P} = (O, \sqsubseteq, \psi)$ *such that* O *is a set of progress values,* $\sqsubseteq$ *is a total order on* O*, and* $\psi : S \to O$ *is a progress mapping where* S *is the set of states.*

The algorithm starts with the initial states as roots for the first sweep (line 7) and proceeds by processing each state in a *least-progress-first order* (line 15). Newly discovered successors are inserted into a queue of unprocessed states (line 28). For each state that is removed from the queue, i.e., the state that is now being processed, one checks if that state is in a different layer than the current layer (line 17). If so, non-persistent states are deleted, and the progress value for the current layer is updated. For every new state discovered as a successor one checks if it was found through a regress edge (line 24). If so, the state is added as a persistent state (line 25).

Running the algorithm on the state space in figure 2.1, it starts with state 1 as root and finds state 2 as a successor. When processing state 2, the algorithm discovers that this state is in a different layer than before, which leads to the deletion of state 1 from memory. Successor states 3 and 4 are discovered and processed, so that state 5 is found. When state 5 is processed, it is in a new layer

Algorithm 2 The sweep-line algorithm

```
1: Set of states R                          ▷ Set of states used as roots for next sweep
2: Set of states N                          ▷ Set of states representing the states in memory
3: Set of states U                          ▷ Set of states that are unprocessed
4: Set of states L                          ▷ Set of states that are in the current layer
5: Set of states P                          ▷ Set of states that are persistent and cannot be deleted

6: procedure SWEEP(s_I)
7:     R ← s_I                              ▷ Set initial states as roots for first sweep
8:     N ← s_I
9:     while R ≠ ∅ do
10:        U ← R
11:        R ← ∅
12:        DELETENONPERSISTENTSTATES()
13:        let ψ(c) be such that c ∈ U and ∀ c' ∈ U : ψ(c) ⊑ ψ(c')
14:        while U ≠ ∅ do
15:            let s be such that s ∈ U and ∀ s' ∈ U : ψ(s) ⊑ ψ(s')
16:            U ← U \ s
17:            if ψ(c) ≠ ψ(s) then
18:                DELETENONPERSISTENTSTATES()
19:                ψ(c) = ψ(s)
20:            L ← L ∪ s
21:            for all (s, t, s') such that s -t-> s' do ▷ Valid transition from s to s'
22:                if (s' ∉ N) then
23:                    N ← N ∪ s'
24:                    if ψ(s) ⊐ ψ(s') then    ▷ s' has lower progress value than s
25:                        P ← P ∪ s'
26:                        R ← R ∪ s'
27:                    else
28:                        U ← U ∪ s'
29: end procedure

30: procedure DELETENONPERSISTENTSTATES(void)
31:     for all s' ∈ L \ P do
32:         N ← N \ s'
33:     L ← ∅
34: end procedure
```

which leads to the deletion of the states 2, 3, and 4. Further, state 1 is found through the regress edge, and thus is marked as persistent. The algorithm must go through the state space once more, but in the second round state 1 is not deleted from memory. State 1 is therefore recognized as a previously visited state when successor states from 5 is computed, and the algorithm terminates.

The pseudo-code in algorithm 2 is inspired by [4], but with an important difference in line 17 where the sign is corrected to $\neq$ instead of $<$. This change was necessary because in [4] one only deleted states when encountering a new layer when making *progress*. However, in the presence of a regress edges, the algorithm described in [4] would not delete the states correctly. This would lead to unnecessary many states in-memory, and early termination. In the corrected pseudo-code, states are deleted when starting to process a new layer, regardless of making *progress* or *regress*.

2.2 Algorithmic Operations

The algorithmic operations needed by the sweep-line method (algorithm 2), and how to implement them efficiently is one of the research questions of this master's thesis.

Since states are explored in a *least-progress-first* order a priority queue of unprocessed states which supports efficient insert (line 28), find-min (line 13, 15) and delete-min (line 16) is needed.

The algorithm also rely upon the information on which states belong to the current layer, and which states are persistent to determine which states can be deleted from memory when entering a new layer. In the procedure of deleting non-persistent states starting at line 30, it uses the two sets, $\mathcal{L}$ and $\mathcal{P}$, to efficiently determine this.

When a state is discovered through a transition, one needs to determine whether we have this state already in memory (line 22) and thus it should not be processed.

The key operations and data structures for the sweep-line method that needs to be efficient are therefore:

- A priority queue for unprocessed states.
- A set with a find method for determining if a state belongs to a set or not.
- Insertion and deletion methods for the representation of which set each state belongs to, e.g., persistent, unprocessed, roots.

These operations are also heavily used for many of the algorithms which is described later in this thesis.

2.3 On-The-Fly Model Checking

One way of model checking a state space for properties is to compute the complete state space in advance and then go through each state afterwards to check the properties. This is called *off-line* state space exploration, where the complete state space is known when the model checking algorithm runs. If one considers a safety property this means that in the case of a initial state does not satisfy the property, the off-line procedure will not immediately recognize this, but continue to compute the rest of the state space. Only when going through the states afterwards, the actual checking of the properties occurs.

Another way to conduct model checking, is to check the properties *on-the-fly*. This means checking of properties is done simultaneously as computing the successors of each state. With on-the-fly model checking one only computes the successors states needed, and if a property is violated it is recognized before the computations of additional successors. One is thus able to report this, and abort the exploration of the rest of the state space.

The sweep-line method is tailored for checking state spaces that does not fit in computer memory, and must therefore rely upon on-the-fly model checking.

Chapter 3

Automata-Based Property Checking

Compared to the standard model verification algorithms the key difference of the sweep-line method is that it deletes states on-the-fly. Therefore, most of the standard verification algorithms are not directly applicable with the sweep-line method. Research have been done on how to model check properties, and providing counter examples, similar to the standard verification algorithms. In this chapter, methods for automata-based property checking with the sweep-line method are explained.

3.1 Standard Automata-Based Checking

In automata-based verification one specifies an automaton that recognizes the undesired behaviours (words) generated by a transition system. Stated in section 1.3, a run for a transition system generates a word by considering the actions performed or states encountered, and all possible runs generates the language of a system. The property holds if the languages of the transition system and the automaton are disjoint. That these languages are disjoint means that no words generated by the transition system are accepting words in the automaton representing the negated property.

To verify a property one performs a parallel composition of a system model, represented as a transition system, and an automaton accepting the *negation* of the desired properties of the system. The negation is used because one tries to find the bad traces of a model that constitute undesired behaviour.

When a product transition system is constructed, one ends up with a composition that holds information on which states and transitions that are reachable in both

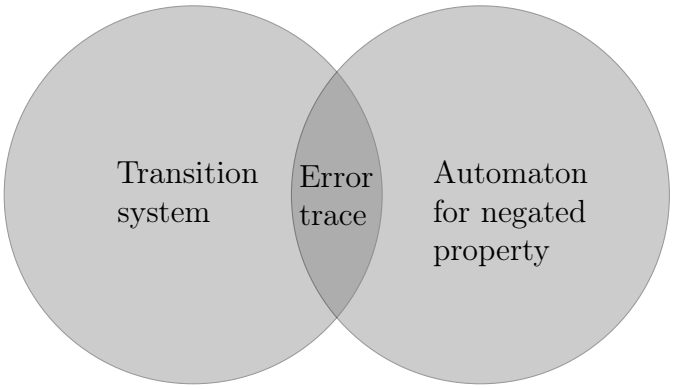


Figure 3.1: Language of a transition system and an automaton.

the transition system and the automata. One can use this to perform a state space traversal on the composition and verify which properties that holds according to the acceptance states in the composition. In the case that one wants to check for a regular safety property, this is equivalent to check that the composition does not have an acceptance state [3, p. 164].

Figure 3.1, shows a transition system that represents a program, and an automaton specifying all words that should not be present in that program. If the two languages of the transition system and automaton intersect, this means that the transition system has unwanted behaviours. The words that lies in this intersection are called error traces.

Figure 1.3 in section 1.3 depicts a transition system of two processes that enters a critical section. To verify that these two processes cannot enter the critical section at the same time, one construct a finite automaton which accepts the negated property, i.e., the automaton will accept an execution has both processes in the critical section simultaneously. Figure 3.2 depicts an automaton starting in the initial state q_0, and will stay in this state as long as both processes are not in the critical state at the same time. If there is a state that is labelled with both crit1 and crit2, the automaton will move to state q_1 which is a *trap state* meaning that regardless of which action occurs it will stay in this state. Figure 3.2 depicts such a trap state on q_1 with a self-loop on the action * which denotes a wild card, i.e., it matches any action performed.

To verify that the regular safety property "only one process should enter the critical section at any time" one performs a parallel composition of the transition system (figure 1.3) and the automaton (figure 3.2). The state space of the parallel composition represents the combined behaviour, and one performs a state space traversal to check that there is no state where the automaton is in the acceptance state. One is only concerned with acceptance state in the automaton because all states in the transition system are treated as acceptance states. The property does

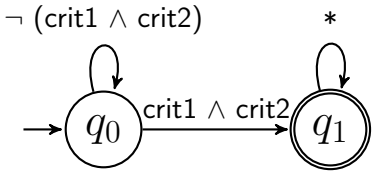


Figure 3.2: Automaton accepting a run if two processes are in a critical section at the same time.

of course not hold in the example as there are no means of restricting the critical section to only one process in the model. Figure 3.3 illustrates the resulting product of the parallel composition. In the verification one wants to prove that no state is labelled q_1 which is the acceptance state of the property automaton. The labelling of the state where both processes are in the critical section, and all states following that state includes q_1. This demonstrates that indeed the property does not hold.

Finite automata can be used when recognizing regular languages. However, some properties have error traces represented by *ω-regular languages*, which are languages over infinite words. Using ω-regular languages one can for instance express that each process in a system should enter the critical section infinitely often. The only way that this property can be violated is if there exists a cycle not containing a process in the critical section. In other words, if the system has an execution path allowing the processes to never enter the critical section.

The conventional approach to on-the-fly model checking of ω-regular language is based on the exploration of the product transition system of a *transition system* and the Büchi automaton that is constructed from the negated LTL formula. The product transition system is searched using a *nested depth-first search* [3, p. 202] to detect an *acceptance cycle*, i.e., a cycle containing an acceptance state [14].

Figure 3.4 depicts the automaton that accepts the *liveness* property that process 1 should enter its critical section infinity often. This property can only be expressed with an automaton that accepts infinite runs (words).

3.2 Parallel Composition

In automata-based verification as described in section 3.1, the concept of taking a transition system representing a system under verification and a non-deteministic finite automaton (NFA) representing the negation of a desired property that we want the system to possess was introduced. In the verification process, one combines the transition system and the NFA to obtain a product of the two. This is called parallel composition. In this section, a more formal description of

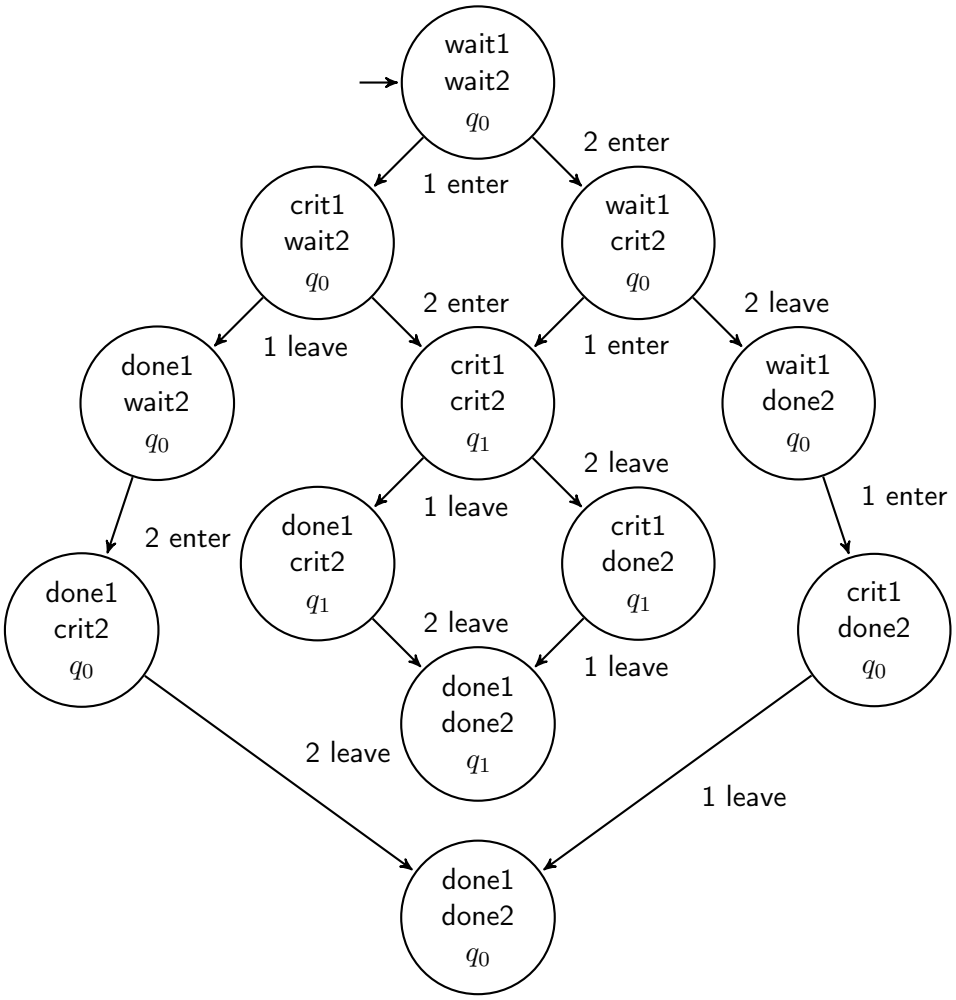


Figure 3.3: Parallel composition of transition system in figure 1.3 and the automaton in figure 3.2.

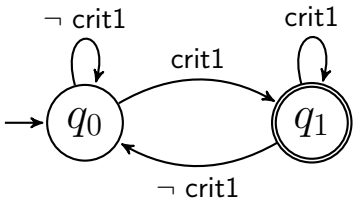


Figure 3.4: Automaton for ω-regular properties.

a parallel composition is presented using definitions from "Principles of Model Checking" [3], and it is explained how it can be done algorithmically with the sweep-line method.

For the transition system the definition by [3, p. 20] is used:

Definition 3.1 (Transition System)
A transition system TS is a tuple (S, Act, $\rightarrow$, I, AP, L) where

- *S is a set of states,*
- *Act is a set of actions,*
- $\longrightarrow \subseteq (S \times Act \times S)$ *is a set of a transition relation,*
- $I \subseteq S$ *is a set of initial states,*
- *AP is a set of atomic propositions, and*
- $L : S \rightarrow 2^{AP}$ *is a labelling function.*

  TS is called finite if S, Act, and AP are finite.

In words, one has a set of states where a subset of them are initial states. The set of actions define the actions possible in the system. The transition relation holds the information on which actions are enabled in each state, and maps the current state together with a destination state according to which action is performed by the system. The set of atomic propositions defines all basic properties the system may possess, and can for instance be linked to the values of variables of the system. The labelling function labels each state with the respective atomic properties that holds in each state.

For an NFA, the definition of [3, p. 149] is used:

Definition 3.2 (Non-deterministic Finite Automaton)
A non-deterministic finite automaton NFA $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, \Sigma, \delta, Q_0, F)$ where

- *Q is a finite set of states,*
- Σ *is a finite alphabet,*
- $\longrightarrow \subseteq (Q \times \Sigma \times Q)$ *is a transition relation,*
- $Q_0 \subseteq Q$ *is a set of initial states, and*
- $F \subseteq Q$ *is a set of accept (or: final) states.*

In words, one has a set of states, where a subset of them are initial states. The alphabet defines which symbols the NFA can read, and the transition relation maps a current state to a set of destination states according to which input symbol is read. The final states are a subset of the set of states define which states that are accepting in the NFA. A word is accepted if it is processed and ends in an acceptance state. The transition relation is altered from [3, p. 149] in order to be consistent with the transition relation used for the transition system.

When performing a parallel composition, one wants the resulting product to express the combined behaviour of the transition system and automaton. This means that when the transition system moves from one state to another, one must move accordingly in the NFA. Conceptually, the process of parallel composition is to perform an action on the transition system and for each action *act* performed, one processes *act* as an input word on the NFA. In addition, one needs to make sure that only those actions can be processed by the NFA in its current state are performed. With the sweep-line method (or other model checking techniques) this parallel composition can be done on-the-fly during the verification process. Informally, the initial state is the *product* state that represents the combination of an initial state in the transition system and an initial state in the NFA. For each move in the product, the new transition relation is restricted to allow only transitions that are enabled in both the transition system and the NFA. Computing initial states and all states possible from the transition relation yields the product of the two, with their combined behaviour. In the verification process, the labelling function of the transition system holds the information on which atomic propositions holds in each state, but when validating a property, one is only interested in when the automaton is in an acceptance state. The labelling function of the product therefore only labels the states with the automaton labels, and one does not combine the labels.

The definition of the product of a transition system and an NFA is defined by [3, p. 162]:

Definition 3.3 (Product of TS and NFA)
Let $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states and $A = (Q, \Sigma, \delta, Q_0, F)$ an NFA with the alphabet $\Sigma = 2^{AP}$ and $Q_0 \cap F = \emptyset$. The product transition system $TS \otimes A$ is defined as follows:

$TS \otimes A = (S', Act, \rightarrow', I', AP', L')$

- $S' = S \times Q$,
- $\rightarrow'$ *is the smallest relation defined by the rule* $\dfrac{s \xrightarrow{\alpha} t \wedge q \xrightarrow{L(t)} p}{\langle s, q\rangle \xrightarrow{\alpha}{}' \langle t, p\rangle}$
- $I' = \{\langle s_0, q\rangle \mid s_0 \in I \wedge \exists q_0 \in Q_0 : q_0 \xrightarrow{L(s_0)} q\}$,
- $AP' = Q$, *and*
- $L' : S \times Q \rightarrow 2^Q$ *is given by* $L'(\langle s, q\rangle) = \{q\}$.

For ω-regular properties, a similar process can be done with a transition system and an non-deterministic Büchi automaton (NBA). An NBA is defined in a simple way as an NFA, but with the key difference that infinite word is accepted if the automaton enters an acceptance state infinite often.

3.3 Safety Property Checking

In its basic form the sweep-line method can only be used for verifying regular safety properties. When verifying regular safety properties, one takes a transition system and a non-deteministic finite automaton (NFA) representing the safety property and perform a parallel composition. As described in section 3.2, this can be done on-the-fly with the sweep-line method. When algorithmically performing a parallel composition, one constructs *composition pairs* (s_m, s_a) where the s_m component represents a state of the model, and the s_a component represent a state in the NFA. Algorithm 3 demonstrates how one can compute the successors of a composition pair. This is done by computing a successor s'_m of the model component, and then one needs to get the NFA successors by retrieving the successor s'_a when reading the predicate that holds in state s'_m, resulting in a composition successor (s'_m, s'_a). Both the model component and the NFA component can have several successors, and one therefore needs to iterate through both to capture all possible successor components. In case the NFA does not have a successor for the current predicate in the model component, one does not construct a successor component. This ensures that the definition of a parallel composition is respected, in that one only captures the combined behaviour of a model and an NFA in the resulting product transition system.

Algorithm 3 Computing successors in on-the-fly parallel composition

```
1: model                                          ▷ Interface describing the model
2: nfa                          ▷ Interface describing the non-deteministic finite automaton
3: S                                              ▷ Set of component successors

4: procedure COMPUTESUCCESSORS(s_m, s_a)
5:     M ← model.getSuccessors(s_m)
6:     for all s'_m ∈ M do
7:         predicate ← model.getPredicateInState(s'_m)
8:         N ← nfa.getSuccessors(s_a, predicate)
9:         for all s'_a ∈ N do
10:            S ← S ∪ (s'_m, s'_a)
       return S
11: end procedure
```

When a parallel composition is performed one ends up with a product transition system. The product transition system represents the combined behaviour of the transition system under evaluation and the NFA representing the property to verify. To evaluate the system, one is interested whether there exists an execution of the transition system that is an accepted by the NFA. In "Principles of Model Checking" it is stated that verifying regular safety properties is reduced to invariant checking [3, p. 162]. This means that one is only interested in checking if there is a state that is reachable from an initial state where the automaton is in an acceptance state. It is therefore sufficient to explore the whole state space, which

the sweep-line method guarantees, and check whether the automaton enters an acceptance state.

3.4 Linear Temporal Logic Property Checking

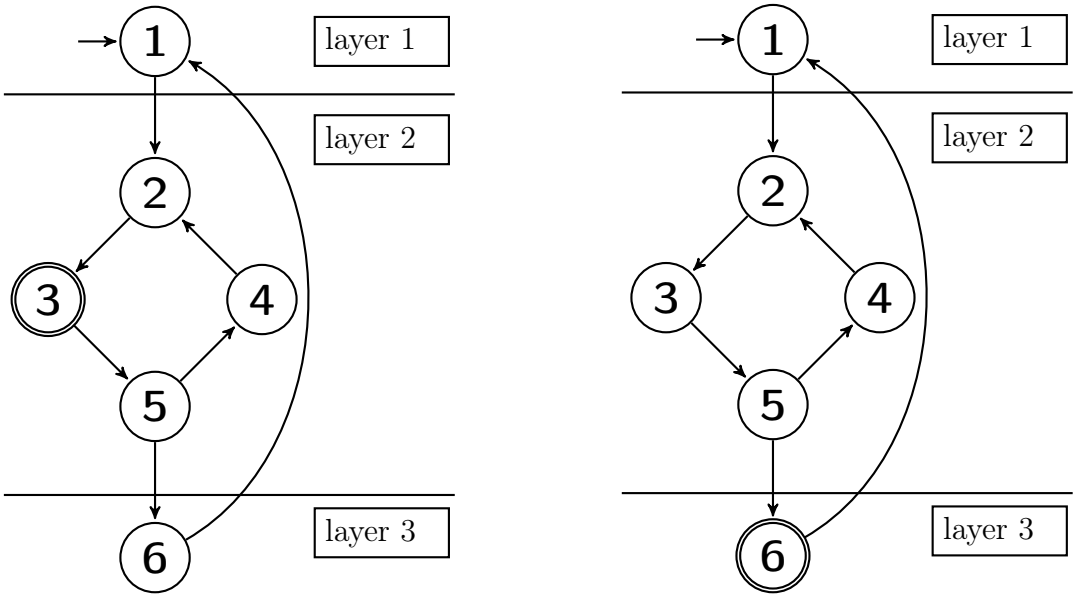
When verifying linear temporal logic (LTL) it is not sufficient to just explore the state space as with safety properties. As explained in section 3.1, properties expressed in LTL are verified by looking for an acceptance cycle. For the sweep-line method these cycles can contain states that span multiple layers and thus have been deleted in the verification process.

To support LTL it was proposed in the paper [15] to split the property checking in two: the detection of a *single layer acceptance cycle (SLAC)* see figure 3.5a, and the detection of a *multi-layer acceptance cycle (MLAC)* see figure 3.5b. In its basic form, the algorithm starts to search for an SLAC, and if no one were found then it continues to search for an MLAC.

When searching for an SLAC, one is only concerned if the set of states that constitutes an acceptance cycle are in the same layer. This means that to verify a property, one can use the standard *nested depth-first search* [3, p. 202] constrained to explore one layer at the time.

When searching for an MLAC, one exploits the fact that a cycle is going through a *persistent state*, or else it would be an SLAC. Therefore, one starts with the set of persistent states that were found when searching for an SLAC, and try to find cycles containing these, where at least one state in the cycle is an acceptance state. In [15] they took the basis of the MAP algorithm [16], that was used to find an acceptance cycle by looking for the *maximal accepting predecessor* of each state, and altered it to compute a *maximal persistent predecessor* for each state. This makes it possible to determine if there is a cycle with an acceptance state in it. The original MAP algorithm was not directly applicable because it would require to hold in memory all acceptance states, and because the sweep-line method deletes states on-the-fly this is not possible. The intuition behind finding the *maximal predecessor* is that all states in a cycle necessarily share a common predecessor, and in order to be well-defined, the maximal predecessor is used.

The augmented MAP algorithm in [15] defines an *mpp* function that intuitively means that if $\text{mpp}(s) = (p,\ b)$ then p is the largest persistent state that can reach s and the boolean value b is true if and only if there is a path from p to s containing an acceptance state. It follows that if a state s has $\text{mpp}(s) = (s,\ \textit{true})$ this means that s can reach itself, i.e., a cycle, and that there is an acceptance state in that cycle. The definition of maximal persistent predecessor below is



(a) Single layer acceptance cycle with states 2-3-5-4-2.

(b) Multi layer acceptance cycle with states 1-2-3-5-6-1.

Figure 3.5: Schematic of different acceptance cycles.

changed to using a transition system, and the notation $s \rightarrow^+ s'$ denotes that there exists a path from s to s' with at least one transition.

Definition 3.4 (Maximal Persistent Predecessor)
Let TS = (S, Act, →, I, AP, L) be a transition system, $P \subseteq S$ be a set of persistent states and $<_S$ be a total order relation on S. The maximal persistent predecessor function $mpp^{\mathcal{P}}_{TS}$: $S \rightarrow \{\perp\} \cup (P \times \{false, true\})$ is defined by:

$$mpp^{\mathcal{P}}_{TS}(s) = \begin{cases} (p, true), & \text{if } R = \{p' \in P \mid p' \rightarrow^+ \ s\}, R \neq \emptyset, \forall p' \in R \setminus \{p\}, p >_S p' \\ & \text{and } \exists a \in A \mid p \rightarrow^+ \ a \rightarrow^+ \ s \\ (p, false), & \text{if } R = \{p' \in P \mid p' \rightarrow^+ \ s\}, R \neq \emptyset, \forall p' \in R \setminus \{p\}, p >_S p' \\ & \text{and } \nexists a \in A \mid p \rightarrow^+ \ a \rightarrow^+ \ s \\ \perp & \text{otherwise} \end{cases}$$

The MLAC algorithm is presented in pseudo code in algorithm 4 and is inspired by [15]. The *-sign is used to denote a wild card, and the sign $\perp$ denotes that the value is unset, i.e., null. The algorithm starts with the set of persistent states $\mathcal{P}$ as mpp candidates, and for all states $p \in \mathcal{P}$ mpp(p) = $(p,$ *false*$)$. Then p propagates its own mpp to its neighbours, which updates their mpp if the mpp propagated is strictly greater (line 37). Then the algorithm adds to the queue of unprocessed the newly encountered states that got their mpp updated (line 38).

These will further propagate their mpp. The core idea is to always propagate the largest mpp so that if there exists a cycle, then this will be found and one ends up with a state s with mpp(s, b) meaning that a cycle has been found. If the cycle contains an acceptance state the boolean b would be updated to true when that acceptance state is encountered (line 12), and therefore when a state s has mpp(s, *true*) an acceptance cycle has been found (line 32).

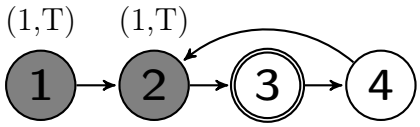
For the total order relation $>_S$, we use the definition from [15], but modified to fit transition systems:

Definition 3.5 (Total Order Relation $>_{mpp}$)
Let TS = (S, Act, →, I, AP, L) be a transition system and $>_S$ be a total order relation on S. We define the total order relation $>_{mpp}$ on $\{\bot\} \cup (S \times \{false, true\})$ as follows:

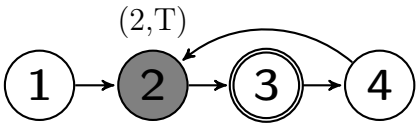
$$m >_{mpp} m' \Leftrightarrow \begin{cases} \quad m = (s,b) \wedge m' = (s',b') \wedge (s >_S s' \vee s = s' \wedge b \wedge \neg b') \\ \vee \; m = (s,b) \wedge m' = \quad \bot \end{cases}$$

When computing the maximal persistent predecessor, an acceptance cycle can be *hidden* by a longer cycle that does not contain an acceptance state. Consider figure 3.6 as a fragment of a larger state space (otherwise state 1 would not be a persistent state). The hiding occurs when a persistent state p propagates its mpp forwards, and another persistent state p' updates its own mpp with mpp(p, b), meaning that if an acceptance cycle exists with mpp(p', b) then this would not be found as depicted in figure 3.6a. Therefore, when a persistent state is processed with an mpp that is larger than itself, i.e., it is another persistent state that is propagated to it, then one must later re-explore this state with its own mpp as there could be a hidden acceptance cycle as depicted in figure 3.6b. All persistent states that have been fully explored that do not meet the requirement, i.e., that its mpp has been propagated to the whole state space but no cycle has been reported, then these can be removed from the mpp candidates as there are no acceptance cycle with that state as maximal persistent predecessor. One can then start a new run of propagating the mpp for the persistent states that are still mpp candidates, to see if an acceptance cycle where hidden by some of the newly removed persistent states. When all the persistent states have been removed as an mpp candiate and no acceptance cycle is found, then the algorithm terminates.

Running algorithm 4 on the state space in figure 3.6 it has the two states 1, 2 in the set of persistent states where state 1 > state 2 (persistent states are coloured grey). Their mpp is set to mpp(1) = (1, false) and mpp(2) = (2, false) (line 12), and all other states s have mpp(s) = ⊥. The mpp of the persistent states



(a) Acceptance cycle is 2-3-4-2 hidden by 1.



(b) 1 is removed from the set of persistent states and acceptance cycle 2-3-4-2 is found.

Figure 3.6: Uncovering of a hidden acceptance cycle.

are shown as a pair above the states in figure 3.6. The algorithm starts with state 1 and propagate the mpp(1, false) to its direct successors (line 24), which is state 2. State 2 updates its mpp because (1, false) $>$ (2, false). It follows that when processing state 2, state 3 updates its mpp to (1, false) and is added to the queue of unprocessed nodes (line 38). Since state 3 is an acceptance state it propagates mpp(1, true) (line 21), which after state 4, reaches state 2. State 1 is then removed as an mpp candidate (line 7), and it begins the same procedure with state 2 with mpp(2) = (2, false). This time the mpp is propagated to states 3 and 4 and is again propagated back to state 2 with mpp(2, true) and an acceptance cycle has been detected (line 32).

Algorithm 4 Sweep-line algorithm for finding multi-layer acceptance cycle

1: **procedure** FINDMLAC($PersistentStates$)
2: $\mathcal{P} \leftarrow PersistentStates$
3: $\mathcal{N} \leftarrow PersistentStates$
4: **while** $\mathcal{P} \neq \emptyset$ **do**
5: $\mathcal{D} \leftarrow \mathcal{P}$
6: MPP()
7: $\mathcal{P} \leftarrow \mathcal{P} \setminus \mathcal{D}$
8: $\mathcal{P} \leftarrow \mathcal{P} \setminus \{s \in \mathcal{P} \mid mpp(s) = (*, false)\}$
9: **end procedure**

10: **procedure** MPP()
11: **for all** $s \in \mathcal{P}$ **do**
12: $mpp(s) \leftarrow (s, s \in A)$
13: $\mathcal{U} \leftarrow P$
14: **while** $\mathcal{U} \neq \emptyset$ **do**
15: $\mathcal{L} \leftarrow \emptyset$
16: let $\psi(c)$ be such that $c \in U$ and $\forall\ c' \in U : \psi(c) \sqsubseteq \psi(c')$
17: $s \leftarrow \varnothing$
18: **while** $\psi(s) = \psi(c)$ **do**
19: let s be such that $s \in U$ and $\forall\ s' \in U : s \sqsubseteq s'$
20: $(p, acc) \leftarrow mpp(s)$
21: $prop \leftarrow (p, acc \vee s \in \mathcal{A})$
22: **if** $s \in \mathcal{P} \wedge p >_S s$ **then**
23: $\mathcal{D} \leftarrow \mathcal{D} \setminus s$
24: VISIT(s, prop)
25: **if** $s \notin \mathcal{P}$ **then**
26: $\mathcal{L} \leftarrow \mathcal{L} \cup s$
27: $\mathcal{N} \leftarrow \mathcal{N} \setminus \mathcal{L}$
28: **end procedure**

29: **procedure** VISIT(S, PROP)
30: **for all** (s, t, s') such that $s \xrightarrow{t} s'$ **do**
31: **if** $prop = (s', true)$ **then**
32: **return** "MLAC found"
33: **if** $s' \notin \mathcal{N}$ **then**
34: $\mathcal{N} \leftarrow \mathcal{N} \cup s'$
35: $mpp(s') \leftarrow \bot$
36: **if** $prop >_{mpp} mpp(s')$ **then**
37: $mpp(s') \leftarrow prop$
38: $\mathcal{U} \leftarrow \mathcal{U} \cup s'$
39: **end procedure**

Chapter 4

Computation Tree Logic Property Checking

As for automata-based model verification algorithms, the key difference with the sweep-line method and standard computation tree logic (CTL) algorithms is the deletion of states on-the-fly. For linear temporal logic (LTL) properties cycle detection is used in the verification. As described in section 3.4, this can be done with the sweep-line method. Properties stated with CTL formulas, are verified by computing the satisfactory set for each sub formula with algorithms depending on the sub formula, using the previously computed information when computing the set for the next part of the sub formula. Several of the algorithms used for computing the satisfactory sets in CTL are recursive. Because of the dependency on each previously computed information and that recursion is not compatible with the least-progress-first traversal of the sweep-line method, CTL properties is still an open research question for the sweep-line method. In this chapter, the standard CTL algorithm is briefly described. We also explain how safety CTL properties can be verified with the sweep-line method.

4.1 Standard Computation Tree Logic Checking

When verifying a CTL property Φ on a transition system, one needs to check if all initial states satisfy Φ. Since Φ can have several sub formulas, one must therefore compute which states that satisfies all sub formulas of Φ first. For the formula EF$\bigcirc$ a (there exists a state where the next state is labelled with a), one must first compute all the states that are labelled with the atomic proposition a, and then one can compute all states that are direct predecessors of the set computed beforehand. In "Principles of Model Checking" it is stated that verification of CTL properties boils down to bottom-up traversal of the *parse tree* of the formula [3, p. 336].

The parse tree of a CTL formula is a tree representing the formula with each sub formula in its own branch. The leaves of the parse tree are atomic propositions that can be computed directly from the labelling function of the transition system. Figure 4.1 depicts the parse tree of the CTL formula $\Phi = \text{AG } a \wedge \text{EF } \neg b$ inspired by [3, p. 338].

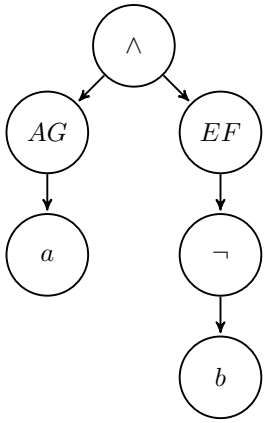


Figure 4.1: Example of a CTL parse tree.

Since mutual exclusion is a safety property, it can be verified with both LTL and CTL. Therefore, as in section 3.1, one can verify the same property on the transition system TS from figure 1.3 in section 1.3 using CTL. The respective CTL formula is: AG $\neg$(crit1 $\wedge$ crit2). Figure 4.2 depicts the parse tree; Ψ_3 and Ψ_4 represents the atomic propositions $crit1$ and $crit2$, respectively. $\Psi_2 = \Psi_3 \wedge \Psi_4$ and $\Psi_1 = \neg\ \Psi_2$. The basic CTL algorithm will start with the leaves Ψ_3, Ψ_4 and find the set satisfying these two properties by only considering the atomic propositions and the labelling function. To compute the set of states that satisfies Ψ_2 one takes the intersection of the two sets previously computed. We then continue to move up the parse tree and compute the next sub formula, Ψ_1, which is the negation of Ψ_2. The set of states that satisfy Ψ_1 is the set of states in the transition system minus the set of states satisfying Ψ_2. When processing the root, Φ, one wants to find out if the set of states satisfying the sub formula Ψ_1 equal to the set of all states in the transition system, i.e,. the property holds if $\forall$ states s of TS : $s \in \Psi_1$.

In figure 4.3, the basic CTL algorithm is visualized. One traverses the parse tree in figure 4.2 in a bottom-up manner and colour the states accordingly to the sub formula. In figure 4.3a all the states that are labelled with both crit1 and crit2 are computed, and the algorithm then computes the next node which is a negation shown in figure 4.3b. Finally, one can compute the root in figure 4.3c. The initial state does not satisfy this formula, thus the transition system does not satisfy the formula. This is consistent with the result when verifying the same property with LTL.

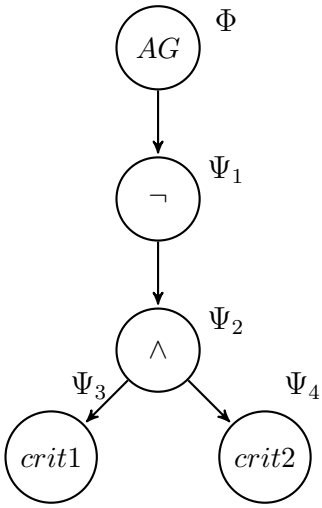


Figure 4.2: CTL parse tree of a safety property.

4.2 Safety Property Checking

CTL is verified in a recursive manner, relying upon predecessors, and is therefore not directly compatible with the exploration order defined by the sweep-line method. In section 4.1, the basic CTL checking algorithm were described, and the main idea is to compute, for all states, the satisfactory set of each sub formula Ψ of the complete formula Φ. Each sub formula in the parse tree may be dependent on the previous information that were computed when processing a sub formula in the parse tree. Since the sweep-line method deletes states when entering a new layer, previously computed information is lost and the standard CTL verification algorithms cannot be used. To the best of our knowledge, property checking CTL formulas with the sweep-line method has not previously been described.

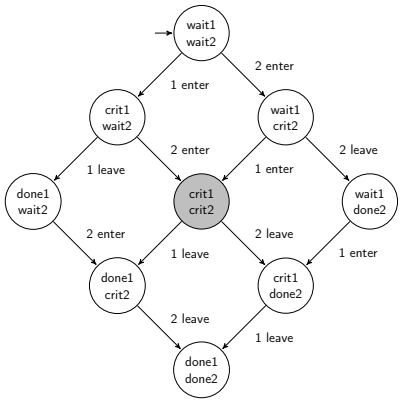
Some CTL properties can be verified rather trivially with the sweep-line method without any limitations. Properties of the form EF Φ (reads *exists finally* Φ) states that it should be possible to reach a state where the state predicate Φ holds from at least one of the initial states in the state space. The way of model verification is to try to prove the negation of the desired property, and report that the property holds if one is unable to prove the negation. Therefore, one wants to prove that there exists no state in the state space where Φ does not hold. Because the sweep-line method guarantees to search the whole state space, checking this property can be done by searching the entire state space and report that the property does not holds if none of the reachable states satisfies Φ, or else report that the property holds.

Properties of the from AG Φ, (reads *all paths global*) are satisfied if all states in the state space that are reachable from the set of initial states satisfy the

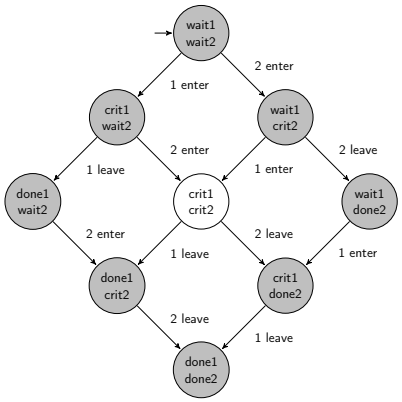
predicate Φ, and is the negated of EF Φ, i.e., $\neg(\text{AG } \Phi) \equiv \text{EF } (\neg\Phi)$. Verifying AG Φ properties is also directly possible with the sweep-line method similarly to the method described above. Since the desired property is that all states satisfies Φ, an error trace is provided if one can find a state that is reachable from an initial state where Φ does not hold, or else report that the property does indeed hold for the state space. We therefore conclude that properties of the form:

- EF Φ
- AG Φ

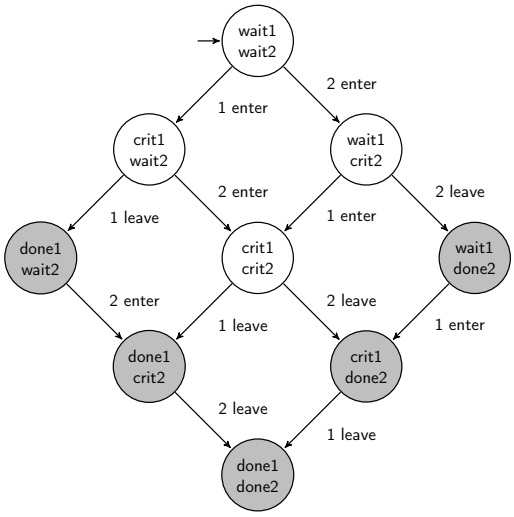
are easy to verify using the sweep-line method without any limitations or sophisticated tailored algorithms. We refer to the basic sweep line algorithm, algorithm 2, in section 2.1 which guarantees to explore the state space. For checking the property one line is appended, to check the predicate at each state according to the desired property specified in the verification.



(a) Set of states satisfying $\Psi'' \equiv$ crit1 $\wedge$ crit2 coloured grey (one state).



(b) Set of states satisfying $\Psi' \equiv \neg$ (crit1 $\wedge$ crit2) coloured grey (eight states).



(c) Set of states satisfying $\Phi \equiv$ AG $\neg$(crit1 $\wedge$ crit2) coloured grey (five states).

Figure 4.3: Visualization of the basic CTL algorithm.

Chapter 5

Computation Tree Logic with Monotonic Progress Measures

In this chapter new methods for checking two key computation tree logic properties when considering monotonic measures is developed. The properties under consideration are EF AG Φ and AG EF Φ. To the best of our knowledge, an algorithm for these two properties has not been developed with the sweep-line method before.

5.1 Key Properties in Computation Tree Logic

Combining the two operators from section 4.2 one can obtain more complex CTL queries:

- AG EF Φ, reads *all global exists finally*
- EF AG Φ, reads *exists finally all global*

Properties of the form AG EF Φ states that from all reachable states there should exist at least one path leading to a state where the predicate Φ holds. Properties of the form EF AG Φ states that at least one valid execution of the model under evaluation leads to a state where all reachable states satisfy the state predicate Φ. These two properties are the negation of each other, in the way that for the first property one wants to check that all states have the possibility to reach a state where some property Φ holds, and in the second one wants to check that there exists an initial state that has the possibility to reach a state where Φ holds in all reachable states. This means that $\neg(\text{AG EF } \Phi) \equiv \text{EF (AG } \neg\Phi)$.

Several papers in explicit-state model checking exploit the use of strongly connected components (SCC) in the verification process, as in for instance [17]. We will

utilize SCCs when checking these properties. In the verification process when checking a property of the form AG EF Φ, one proves that the property does not hold if there exist a state that cannot reach a state where Φ holds. We therefore need to compute the reachability for each state, and check if a state is unable to reach another state satisfying Φ. For the computation of reachability, *terminal* strongly connected components are exploited, as depicted in figure 5.1.

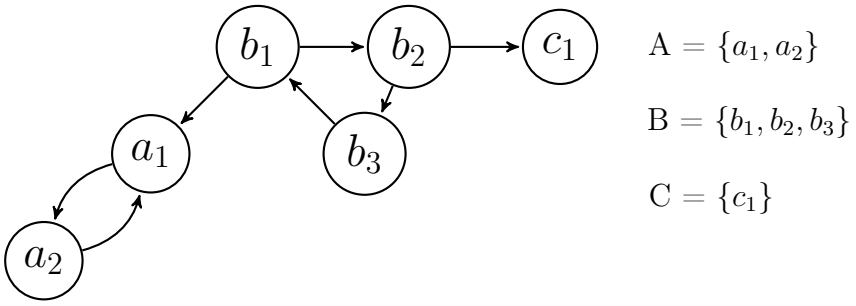


Figure 5.1: The reachability of the states in the terminal strongly connected components A and C are limited to their respective component. The reachability of the states in the B component is the complete state space.

When checking AG EF Φ in the context of the transition system, this means that one must find a terminal strongly connected component where no states in the SCC satisfies the predicate Φ. This also is true for a trivial SCC, i.e., a single state with no out edges. In an SCC, each state can reach every other state in the component, and if in addition the SCC is terminal, then there are no out edges of the SCC and the reachability of each node in the SCC are limited to that SCC. Therefore, if such a terminal SCC exists where no states in the SCC satisfies the state predicate Φ then the property AG EF Φ does not hold.

Definition 5.1 (Strongly Connected Component)
Let TS = (S, Act, $\rightarrow$, I, AP, L) be a transition system. A strongly connected component in TS is a maximal set of states $S' \subseteq S$ such that $\forall s, s' \in S' : s \rightarrow^* s'$. A strongly connected component scc is terminal iff $\forall s \in scc, \forall s' \in S : s \rightarrow^* s' \Rightarrow s' \in scc$.

Stated in words, all states in a strongly connected component can reach each other and an SCC is called terminal if no states in the component can reach any states not belonging to that same component. It follows that a single state with no valid transitions is a terminal strongly connected component.

Since properties of the form EF AG Φ are the negation of AG EF Φ described above, one can in a similar way exploit terminal strongly connected components of the state space when checking these properties. EF AG Φ properties state that there should exist at least one execution path from an initial state to a state that has its reachability limited to only states that satisfy the predicate Φ. We

want to compute the reachability in a similar way as above and conclude that the property is violated if there exists no terminal SCC where all states in the state space satisfy Φ.

We recall that the sweep-line method splits the state space into several layers, processing one layer at a time, and then deletes the non-persistent states from memory. Strongly connected components are not necessarily confined to one single layer, but can be composed of states with different progress values. Therefore, the computation of SCCs is not directly compatible with the sweep-line method.

When limited to only state spaces with *monotonic* progress measures, these properties can be verified. We recall that in definition 2.1, the sweep-line assigns a progress value to all states in a transition system. We call a progress measure *monotonic* if the transition system never makes regress.

Definition 5.2 (Monotonic Progress Measure)
Let $\mathcal{P} = (O, \sqsubseteq, \psi)$ be a progress measure over a set of states S. $\mathcal{P}$ is monotonic if $s, s' \in S$ and $s \rightarrow^* s'$ then $\forall s, s' : \psi(s) \leq \psi(s')$. Otherwise, $\mathcal{P}$ is non-monotonic.

Stated in words, every path in a state space with monotonic measure will lead to a state in the same progress layer or a state in a higher progress layer.

Lemma 5.1
Strongly connected components are confined to a single layer when considering state spaces with monotonic progress measures.

Proof. By definition, a state space with monotonic progress measure will never make regress, and thus have no regress edges. Consider a strongly connected component with state s and state s' in two different layers. If we choose state s to be in the lowest layer of the two, then by definition, this would mean that there is an execution path from state s to state s' making progress, and an execution path from state s' to state s making regress through a regress edge. This can of course not be, and thus no such strongly connected component can exists. □

We exploit lemma 5.1 when verifying the properties stated above. The idea is to explore one layer at a time and compute the strongly connected components for that layer. We therefore need to augment a standard SCC algorithm to only search for strongly connected components in one layer at a time. In addition to find all SCCs of a layer, the augmented SCC algorithm must differentiate terminal SCC and non-terminal SCC as we are only interested in those SCCs that are terminal. To sum up, the main idea of property checking of the CTL formulas:

- AG EF Φ
- EF AG Φ

One must find for each layer the terminal strongly connected components, and check these for the respective properties to be verified according to lemma 5.2.

Lemma 5.2
Let scc_T be a the set of terminal strongly connected components.

- *AG EF Φ holds iff $\forall scc \in scc_T \; \exists s \in scc$ such that $\Phi(s)$ holds*
- *EF AG Φ holds iff $\exists scc \in scc_T$ such that $\forall s \in scc : \Phi(s)$ holds*

Since we limit ourselves to monotonic progress measures this can be done with the sweep-line method.

5.2 Computing Strongly Connected Components

For the computation of strongly connected components (SCC) one can use Tarjan's algorithm [18]. It is a linear time algorithm based upon depth-first search (DFS), and is the "arguably most efficient approach for finding SCCs sequentially" [19]. It is also preferred over Kosaraju-Sharir [20] because it does not require to transpose the edges in the state space [19].

Tarjan's algorithm is based on a recursive DFS search that produces a DFS tree. SCCs form subtrees of the DFS tree [21]. Every state encountered is assigned two values: *disc* and *low*. The disc value represents when a state was discovered in the DFS search, and the low value represents for a state s the lowest disc value of a state s' that is reachable from s. Exactly one state in each SCC will have its disc value equal to its low value, and this is called the *head* of an SCC. States visited are stored on a stack in the order they are visited and are only popped from the stack when their respective SCC has been fully explored. This means that when a head of a SCC is found, that state and all states above it on the stack are in the same SCC.

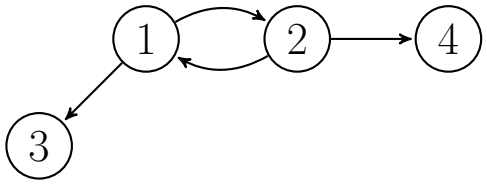


Figure 5.2: Example state space used to demonstrate Tarjan's SCC algorithm.

Figure 5.3 illustrates a run of Tarjan's SCC algorithm. The state space figure 5.2 is considered and the algorithm starts in state 1. Over each state is a (disc, low) pair denoting its disc and low value respectively. The sign $\perp$ denotes that the

value is unset. We write *off* above a state when it is fully explored, and therefore popped off the call stack.

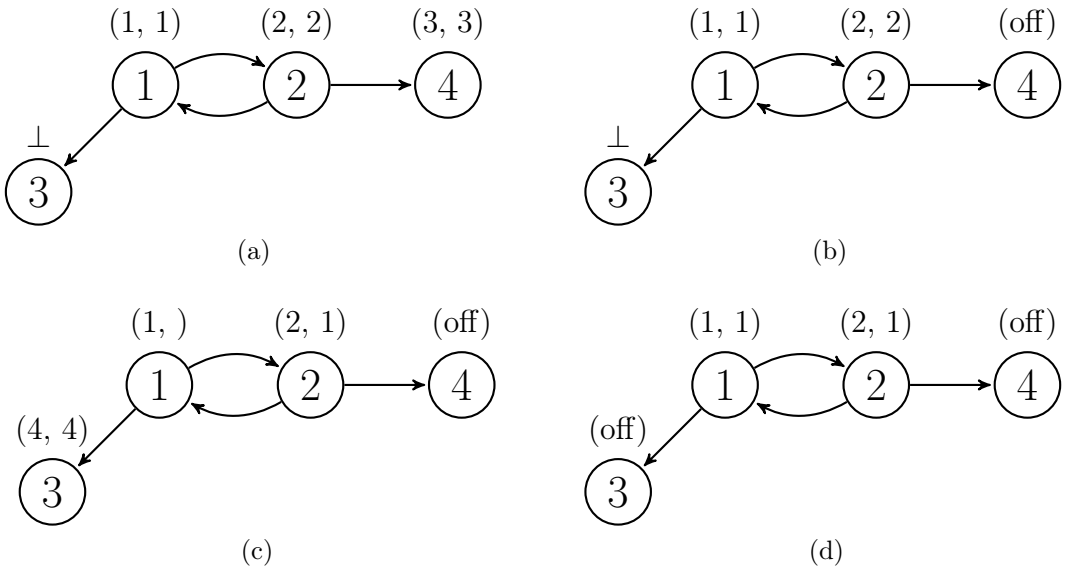


Figure 5.3: Example of Tarjan's SCC algorithm.

In figure 5.3a, state 1 will get disc and low value 1 (line 8), and successor state 2 is discovered. State 2 is discovered for the first time, and the algorithm is recursively called with state 2 (line 11). State 2 receives the disc and low value of 2, before discovering state 3 as a successor. State 3 is discovered for the first time, so a new recursive call is made. State 3 receives disc and low value 3. Since state 3 does not have any successors, the algorithm move to line 16. Its low and disc value are equal meaning that an SCC is found containing state 3 so it is popped from the stack and the recursive call with state 3 returns as depicted in figure 5.3b. The algorithm then continues (figure 5.3c) to go through the remaining successors of state 2. It finds that state 1 is a successor and that state 1 is already on the stack (line 14). The low value of state 2 is therefore updated to be the disc value of state 1 which is 1 (line 15), before the recursive call with state 2 is returned. The state 4 is discovered as a successor of state 1, and in the same way as with state 3 its low and disc values are equal and therefore it is recognized as a strongly connected component and removed from the stack. All successors of state 1 are explored, and in line 16 the algorithm finds that state 1 has equal low and disc value. Since state 2 is still on the stack the algorithm reports that state 1 and state 2 is a strongly connected component as shown in figure 5.3d.

In algorithm 5 is the pseudo-code inspired from [19]. A regular stack data structure is used, with the added function *onstack(s)* that returns true if s is on the stack.

Algorithm 5 Tarjan's algorithm for computing SCC

```
1: time ← 0                                          ▷ Counter representing the time
2: disc          ▷ Data structure to keep track of the discovery time of each state
3: low    ▷ Data structure to keep track of the state with minimum disc time that is
   reachable from each state
4: S                          ▷ Stack representing the current depth-first search stack

5: procedure ComputeScc)(s)
6:     time ← time + 1
7:     low(s) ← time
8:     disc(s) ← time
9:     S.push(s)
10:    for all (s, t, s') such that s --t--> s' do
11:        if disc(s') = ⊥ then
12:            ComputeScc(s')
13:            low(s) ← min(low(s), low(s'))
14:        else if S.onStack(s') then
15:            low(s) ← min(low(s), disc(s'))
16:    if low(s) = disc(s) then                          ▷ head of an scc found
17:        s' ← S.pop()
18:        while disc(s') ≥ disc(s) do
19:            s' ← S.pop()
20: end procedure
```

5.3 Monotonic Algorithm for the Sweep-Line Method

In algorithm 6, the pseudo-code for finding terminal strongly connected components (SCC) with the sweep-line method is presented. It relies upon Tarjan's algorithm for finding strongly connected components. Algorithm 6 is influenced by a standard Tarjan's SCC implementation found in [21].

The standard algorithm relies upon having the whole state space in memory and cannot directly be used with the sweep-line method. In order to be compliant with the sweep-line method, the depth-first search (DFS) is limited to a single layer at a time. The states that are found as a successor in a higher layer are stored in the set $\mathcal{R}$, and will be processed upon entering their respective layers. In addition to finding the SCCs, a check whether the SCCs are terminal is needed.

In algorithm 6, the procedure FindTerminalScc starts the exploration of each layer with a call to the procedure LimitedDfs, and deletes the states from the previously explored layer. The procedure LimitedDfs is a depth-first search (DFS) exploration limited to a single layer at a time. If the DFS encounters a state that belongs to a higher layer than the one currently being processed processing (line

30), it will insert that state to the set of roots and not explore any successors, which is done with the *continue* keyword (line 32) which means that the rest of the for loop is skipped, and it will start with the next iteration. We also note that in line 28 we can detect and report if the state space is not monotonic. When a complete SCC is found in line 36, the method *extractAbove* is called on the stack. This will pop and return the state s it is called with and every state above s (line 37) to retrieve the SCC. Then a check to see if the SCC is terminal is needed in line 38. Checking if a component is terminal can be done by starting a new exploration in one of the states, and check that all encountered states belongs to the component. When a terminal strongly connected component is discovered, one can check it against the desired property in line 39. This procedure must check all state predicates in the SCC and is dependent on the property. For the property AG EF Φ, an error trace is reported if none of the states in the SCC satisfies the state predicate. For the property EF AG Φ, an error trace is reported if at least a state in the SCC does not satisfy the state predicate Φ. Details regarding providing an error trace is discussed in chapter 6.

Algorithm 6 Sweep-line algorithm utilizing terminal SCCs

1: $time \leftarrow 0$ ▷ Counter representing the time
2: $disc$ ▷ Data structure to keep track of the discovery time of each state
3: low ▷ Data structure to keep track of the state with minimum disc time that is reachable from each state
4: $\mathcal{S}$ ▷ Stack representing the current depth-first search stack
5: $\mathcal{L}$ ▷ set of states in layer
6: $\mathcal{N}$ ▷ set of states in memory
7: $\mathcal{R}$ ▷ set of states used as roots for the next sweep

8: **procedure** findTerminalScc()
9: $\mathcal{R} \leftarrow s_I$ ▷ Set initial states as roots for first sweep
10: let $\psi(c)$ be such that $c \in \mathcal{R}$ and $\forall\ c' \in \mathcal{R} : \psi(c) \sqsubseteq \psi(c')$
11: **while** $\mathcal{R} \neq \emptyset$ **do**
12: let s be such that $s \in \mathcal{R}$ and $\forall\ s' \in \mathcal{R} : \psi(s) \sqsubseteq \psi(s')$
13: **if** $\psi(c) < \psi(s)$ **then**
14: deleteNonPeristentStates()
15: $\psi(c) \leftarrow \psi(s)$
16: $\mathcal{S} \leftarrow \emptyset$
17: $\mathcal{L} \leftarrow s$
18: **if** $s \notin disc$ **then**
19: LimitedDfs(s, $\psi(c)$)
20: **end procedure**

21: **procedure** LimitedDfs(s, progressValue)
22: $time \leftarrow time + 1$
23: $disc(s) \leftarrow low(s) \leftarrow time$
24: $\mathcal{S}$.push(s)
25: **for all** (s, t, s') such that $s \xrightarrow{t} s'$ **do** ▷ Valid transition from s to s'
26: **if** $s' \notin \mathcal{N}$ **then**
27: $\mathcal{N} \leftarrow \mathcal{N} \cup s'$
28: **if** $progressValue > \psi(s')$ **then**
29: **report** non-monotonic progress measure detected
30: **if** $progressValue < \psi(s')$ **then** ▷ State s' is in a higher layer
31: $\mathcal{R} \leftarrow \mathcal{R} \cup s'$
32: **continue** ▷ Continue with the next iteration of the loop
LimitedDfs(s', progressValue)
33: $low(s) \leftarrow min(low(s), low(s'))$
34: **if** $\mathcal{S}$.OnStack(s') **then**
35: $low(s) \leftarrow min(low(s), disc(s'))$
36: **if** $low(s) = disc(s)$ **then** ▷ Strongly connected component found
37: $scc \leftarrow \mathcal{S}$.extractAbove($s$)
38: **if** isSccTerminal(scc) **then**
39: CheckProperty(scc)
40: **end procedure**

Chapter 6

Error Traces with the Sweep-Line Method

When a property is violated, an important feature of model checking algorithms is to provide an error trace, constituting a counter example demonstrating why the property does not hold. This counter example is used to understand the error, so that one can correct the model. Verification algorithms that rely upon depth-first search (DFS) or breadth-first search (BFS) with the complete state space in-memory, can backtrack with a predecessor relation from where the property were violated to an initial state. Since the sweep-line method only has one layer in memory at any time and delete non-persistent states, one cannot use the trivial backtrack technique because this information may have been deleted.

6.1 Traces for Safety Properties

When checking for regular safety properties one is only interested in a single state where the property to be verified does not hold, and thus the error trace of interest is a simple path from an initial state to the state violating the property. One proposed method to obtain this trace is to store a tree to external storage [22].

To obtain an error trace, one can extend the basic algorithm so that the initial states are written to external storage, and for each new state that is encountered, it is written to external storage together with its immediate predecessor. In listing 6.2 an example of a file written to external storage is presented. When writing each new state to external storage with its predecessor one obtains a tree on external storage with information of the path in which each state is encountered. Because the sweep-line method is deleting states on-the-fly and potentially must re-explore parts of the state space the same states can be re-discovered several

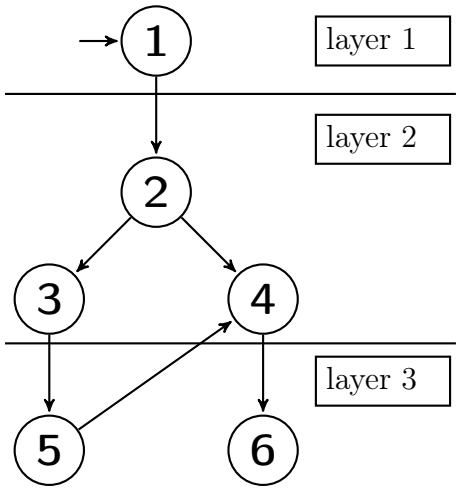


Figure 6.1: Schematic of a state space.

times, and thus be written to external storage several times where its predecessor may vary. When a property is violated one can backtrack to the initial state by searching backwards from the state where the violation is recognized to the initial state using the predecessors written to external storage. In the case a state has been re-discovered several times one is free to choose either predecessor when backtracking. Searching the file in external storage for the predecessor from top to bottom (line 1 to end of file) will yield the immediate predecessor when the state was discovered the first time. When the sweep-line method is traversing the state space, one is only appending to the file on external storage and does not require any searches on external storage. When reporting the error trace one search on external storage is required for each state on the external storage.

Listing 6.1: Output written to external storage for safety error trace.

```
1 1 has direct predecessor: -1
2 2 has direct predecessor: 1
3 3 has direct predecessor: 2
4 4 has direct predecessor: 2
5 5 has direct predecessor: 3
6 6 has direct predecessor: 4
7 4 has direct predecessor: 5
```

The output in listing 6.2 is a result of considering the state space in figure 6.1 constructed with the assumption that whenever there is a choice, the state with lowest numbering is processed first, i.e., after state 2, state 3 is processed before state 4. An illustration of the tree written to external memory is depicted in figure 6.2. The algorithm writes the initial state (state 1) to external storage at the start of the algorithm with a predecessor indicating that this is an initial state,

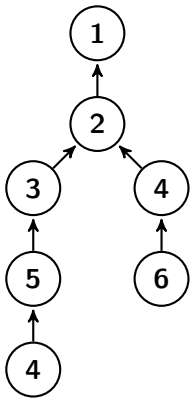


Figure 6.2: Visualization of tree written to disk.

for example -1. When the sweep-line method explores the state space it writes each newly encountered state to external storage together with its predecessor. State 4 it is written to disk two times, but following either predecessor (state 2 or state 5), one will end up with the initial state. If the verification process find that a predicate is violated in state 6 one can follow the tree in figure 6.2 and obtain the reversed trace (6, 4, 2, 1). This can be reversed again and presented as an error trace. This error trace is shown in listing 6.2. From figure 6.1 one sees that this error trace constitutes a path from the initial state 1 to state 6 that violated the property.

Listing 6.2: Safety error trace.

```
1 start of path -> 1 -> 2 -> 4 -> 6 -> end of path
```

The listing 6.2 is rather verbose, and for an implementation one would simply write a (state, predecessor) pair to external storage, i.e., (1, -1) instead of "1 has direct predecessor: -1". The algorithm presenting the trace is shown in algorithm 7. The ordering $\sqsubseteq$ is defined as $(s, pre) \sqsubseteq (s', pre')$ if (s', pre') is the bottommost entry in the file.

Stated in words one starts with the state one is interested in obtaining an error trace to, and add it to the stack which represents the error trace. The algorithm then find its direct predecessor stored on external storage by retrieving the pair (s, pre) where s is the state of interest, and pre is its predecessor. While the predecessor pre of s is not an initial state, one continue to find the predecessor pair on external storage with the new direct predecessor and push s to the stack. When backtracked to an initial state, the algorithm return the revered stack as an error trace.

Algorithm 7 Algorithm for providing safety trace

1: $\mathcal{S}$ ▷ Set of pairs with the direct predecessor relation previously described
2: $\mathcal{L}$ ▷ Stack to provide trace
3: Define $finalPredecessor$ -1

4: **procedure** REPORTTRACE(state)
5: $curr \leftarrow state$
6: **while** $curr \neq finalPredecessor$ **do**
7: $\mathcal{L}$.push($curr$)
8: let (s, pre) be such that $(s, pre) \in \mathcal{S} \wedge s = curr$ and $\forall\ (s', pre') \in \mathcal{S} \wedge s' = curr : (s, pre) \sqsubseteq (s', pre')$
9: $curr \leftarrow pre$

10: **return** reverse($\mathcal{L}$)
11: **end procedure**

6.2 Traces for Linear Temporal Logic

When checking for properties expressed by linear temporal logic (LTL) one is interested in an acceptance cycle. To provide an error trace in case an LTL property is violated, one thus need to report this cycle and a trace from an initial state to a state in the cycle. The paper on how to check LTL properties [15] with the sweep-line method does not describe how to provide an error trace, and to the best of our knowledge, an algorithm for providing such a trace has not previously been described.

In the case of a single layer acceptance cycle then the cycle is confined to a single layer which means that all states in an acceptance cycle are in-memory at the same time. It is therefore trivial to use the techniques based on the standard nested depth-first search [3, p. 202] and through predecessor relations one can obtain a trace for the cycle. With this, and in combination with the method described in section 6.1 one can obtain a trace from an initial state and to a state in the cycle.

We now describe how to provide an error trace for multi-layer acceptance cycle (MLAC), with the following notation: Persistent states are named p, the current state processed by algorithm 4 in line 24 is named s, and the immediate successors of s in line 30 are denoted s'. A state is named s'' if it is a successor of s, but not a direct successor. If the mpp function (described in section 3.4) of a state is denoted with a boolean b, and nothing else is stated, then b can be either true or false.

In order to provide a trace for an MLAC, we note that the algorithm described in

section 3.4 guarantees to report an acceptance cycle if one exists. If the algorithm reports an acceptance cycle going through persistent state p then mpp(p) = (p, *true*). This is the result of persistent state p propagating and updating mpp = (p, b) to states where a subset of them constitutes a cycle containing state p and where at least one state in that cycle is an acceptance state that updates b to true.

Figure 6.3 depicts two persistent states 1 and 3 with $1 > 3$, and state 8 is an acceptance state (persistent states are coloured in grey, and that acceptance states are drawn with two circles). After a run of algorithm 4, state 1 have propagated its mpp to all states in the state space. When processing state 4, this is discovered and an acceptance cycle is reported.

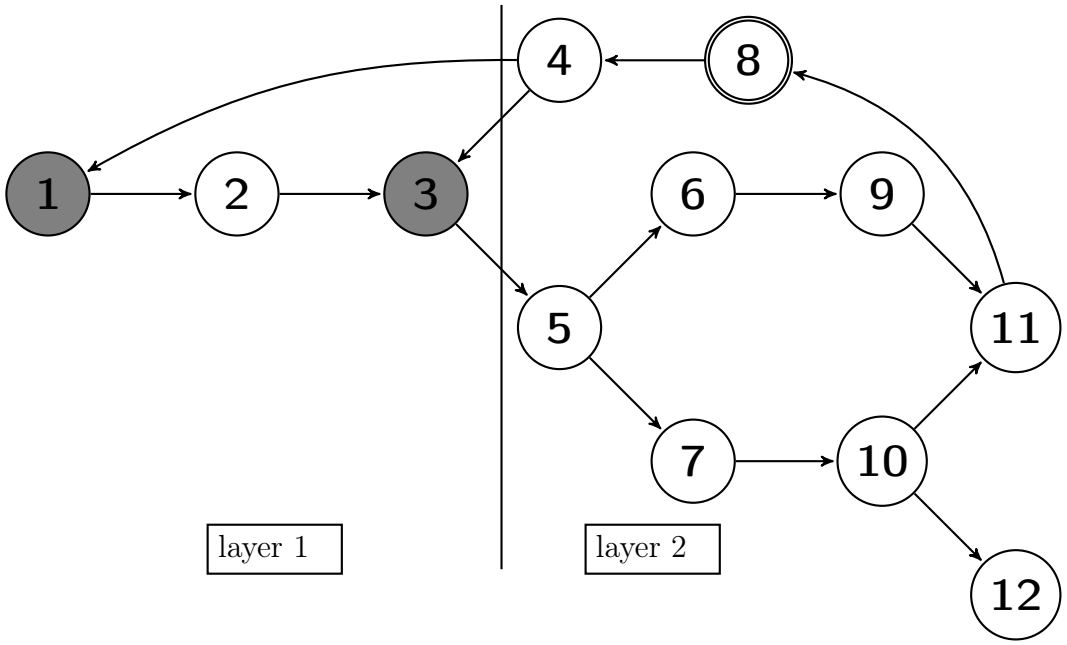


Figure 6.3: Trace example for MLAC.

The key to obtaining error traces with the sweep-line method for LTL is the observation in lemma 6.1 that follows from algorithm 3.4. The lemma holds because if an acceptance cycle is found, the algorithm will terminate, and not explore the possibility for several acceptance cycles.

Lemma 6.1
When an acceptance cycle is found, there exists exactly one persistent state p with mpp(p) = (p, true).

This leads to the following theorem when an acceptance cycle is detected in a persistent state p with mpp(p) = (p, *true*):

Theorem 6.1
Given that an acceptance cycle is found in the persistent state p, an error trace can be provided by backtracking the predecessor relationship of the state s that updated the mpp of state s' by only considering the direct predecessor that updated the mpp of state s' the last time, with mpp = (p, b).

Proof. At least one state which propagated mpp = $(p,\ b)$ is an acceptance state, or else an acceptance cycle would not be reported. It further follows from the algorithm that all propagations of mpp from state s to state s' only occur when state s' is a direct successor of state s.

When state s updates the mpp of state s' with mpp = $(p,\ b)$ there are two possibilities:

1. State s' is not part of any acceptance cycle
2. State s' is part of at least one acceptance cycle

The first case is trivial, since one are not interested in any path starting in state s' and no backtracking from state p would lead to state s'.

If there is only one acceptance cycle, it is also trivial that by following the predecessor relation from state p, one will continue until state p is reached for the second time, because for every state s' only one state s has updated the mpp of state s' with mpp = $(p,\ b)$.

There are two possibilities leading to several acceptance cycles where all states in these cycles have mpp = $(p,\ b)$:

1. There exist more than one path that starts in s, and ends in state s'' where all states including state s and state s'' has mpp = $(p,\ b)$.
2. State s updates the mpp of state s' from mpp(s') = $(p,\ \mathit{false})$ to mpp(s') = $(p,\ \mathit{true})$

In the first case either paths can be reported as part of the error trace, and thus it suffices to choose the one that were updated from s the last time. In the second case this would lead to the cycle being reported not including p, but it is still a cycle and since this can only happen when changing b from false to true it follows that this cycle contains an acceptance state, i.e., an acceptance cycle. □

To illustrate some of the cases described above we consider figure 6.3. State 12 is not part of any acceptance cycle, and that one would not be interested in any path starting in this state. For the case of multiple paths, both paths between state 5 and state 11 can be reported as part of the error trace, and one can choose which one to report as they are both correct. The special case where the cycle reported does not contain the state p in which the acceptance cycle was reported

by the LTL algorithm arises when state 4 is processed and updates the mpp of state 3 from mpp(3) = (1, false) to mpp(3) = (1, true). It follows that the last state that updates the mpp of state 3 is changed from state 2 to state 4.

One can therefore start in the persistent state p and backtrack the predecessor relation and look for which state s that updated mpp = (p, b) to p, and continue to look for the state that updated the mpp of state s until the predecessor relation yields a state that is already reported in the error trace, i.e., a cycle is fully detected.

To provide the error trace, algorithm 4 must be augmented to keep track of which states s updated the mpp of states s'. One must therefore append a line in the procedure visit(s, prop) (line 29) so that for each state s' that updates its mpp based on the prop from state s in line 37, the pair (s', s) is written to external storage indicating that state s updated the mpp of state s'.

Running the augmented algorithm on the state space in figure 6.3, it would produce the output in listing 6.3. Whenever the algorithm had a choice between two nodes, the one with the largest value in figure 6.3 would be processed first.

Listing 6.3: Output written to external storage with the augmented algorithm on the state space in figure 6.3.

```
2 updated by: 1
3 updated by: 2
5 updated by: 3
7 updated by: 5
6 updated by: 5
7 updated by: 10
9 updated by: 6
12 updated by: 10
11 updated by: 10
11 updated by: 9
8 updated by: 11
4 updated by: 8
3 updated by: 4
1 updated by: 4
```

To report the acceptance cycle found by the algorithm, one must start with the state at which the cycle was discovered and follow the direct predecessor relation that is stored on external storage. Since one must find the predecessor that updated the mpp of a state the last time one needs to find the bottommost entry in the file. We continue to find the predecessors until a cycle is discovered.

Listing 6.4 shows the error trace of the cycle reported. To provide a complete error trace, one must provide an error trace from an initial state, to one of the

states in the cycle. This can be done as described in section 6.1.

Listing 6.4: Error trace provided for cycle.

```
start of cycle -> 4 -> 3 -> 5 -> 6 -> 9 -> 11 -> 8 -> 4 -> end of cycle
```

In algorithm 8, the pseudo code to obtain the error trace is presented. The ordering $\sqsubseteq$ is defined as $(s, pre) \sqsubseteq (s', pre')$ if (s', pre') is the bottommost entry in the file.

Algorithm 8 Algorithm for providing acceptance cycle trace

$\mathcal{S}$ ▷ Set of pairs with the update relation previously described
$\mathcal{L}$ ▷ Stack to provide trace

procedure FINDCYCLE(STATE)
    $curr \leftarrow state$
    $pre \leftarrow \varnothing$
    $\mathcal{L}$.push($curr$)
    while $\neg\mathcal{L}$.onstack(pre) **do**
        let (s, pre) be such that $(s, pre) \in \mathcal{S} \wedge s = curr$ and $\forall\ (s', pre') \in \mathcal{S} \wedge s' = curr : (s', pre') \sqsubseteq (s, pre)$
        $curr \leftarrow pre$
        $\mathcal{L}$.push($curr$)

    REPORTTRACE($\mathcal{L}$)
end procedure

procedure REPORTTRACE(stack)
    $first \leftarrow stack$.top()
    $stack$.pop()
    $current \leftarrow \varnothing$
    while $current \neq first$ **do**
        $current \leftarrow stack$.top()
        $stack$.pop()
    return $\mathcal{L}$
end procedure

Stated in words, the algorithm starts with the state that we want to find a cycle from and push it to a stack, used for representing the error trace. As long as we do not have a cycle, we continue to find the bottommost entry on external storage, of which state pre that updated the mpp of the current state. When pre is found, it is pushed the stack. We have a special case if the cycle reported does not contain the state p, that the acceptance cycle was detected on by the LTL

algorithm. This leads to the error trace reported would contain a prefix from p to the cycle. In the procedure *reportTrace*, this prefix is removed and only the states in the acceptance cycle are reported.

6.3 Traces for Computation Tree Logic

Providing error traces for the computation tree logic (CTL) properties developed in this thesis is, in the same way as providing error traces for single layer acceptance cycles, simple.

When limited to state spaces with monotonic progress measures the strongly connected components will be confined to a single layer and thus be in-memory at the same time. We need to report the terminal strongly connected component in addition to a trace from an initial state to a state in the component. The trace from the initial state to the component can be reported as a safety trace with the method described in section 6.1.

Therefore one needs to augment the CTL algorithm to write a tree to external memory, with the information of each new discovered state in addition to its parent and report the strongly connected component which violates the property. In algorithm 6 (section 5.3), in line 39 we have all states in a component in a container when checking if that component violates the property verified. In case an error trace should be reported one can simply report all states in the component in addition to a path from an initial state to a state in the component. Listing 6.5 is an example where a trace is reported for the state space in figure 6.4.

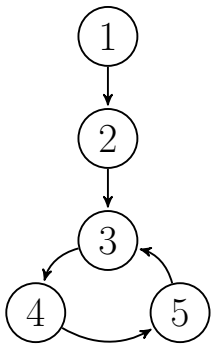


Figure 6.4: State space with a terminal strongly connected component.

Listing 6.5: Error trace example for CTL.

```
1 start of path -> 1 -> 2 -> 3 -> end of path
2 start of component -> 3 -> 4 -> 5 -> end of component
```

Chapter 7

Implementation

The sweep-line method has been implemented in the C++ 14 programming language [23, 24] and results in a console application that can be used to verify properties, and provide error traces in case a property is violated.

7.1 Software Architecture

From the research questions, a unified implementation with well-defined interfaces for the sweep-line method is in focus. To achieve this, we want to extract the core of the sweep-line method and express through interfaces the components the sweep-line method must rely upon in order to conduct model checking. Making these interfaces explicit makes it possible to connect or utilize other tools for expressing the system model or the automaton used for verification. This also makes it possible to integrate our implementation into already existing explicit-state model checkers that do not have the algorithms implemented in this thesis.

The sweep-line method must rely upon a system model, which is depicted in figure 7.1. The model interface is responsible for providing information about the initial state, successors, providing a progress value for a given state and providing the state predicates (atomic propositions) that hold in a state.

When doing automaton-based property checking, one has a model of the software to test, and an automaton describing the negation of the property one wants to verify. The job of the sweep-line method is to do a parallel composition of a model and an automaton and check that they do not share any behaviour, i.e., that the system does not possesses any error traces. The responsibilities of the automaton interface are depicted in figure 7.2 and must provide information about the initial state, the successors of a state given an action to perform, and answer whether a state is an acceptance state in the automaton.

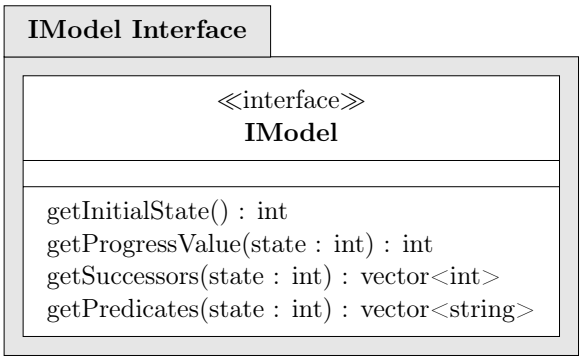


Figure 7.1: Interface for the system model.

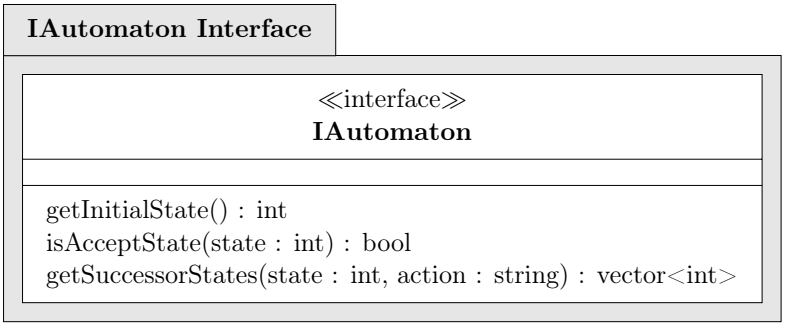


Figure 7.2: Interface for the automaton.

In computation tree logic (CTL) model checking we rely upon an interface describing the CTL formula. This interface is responsible for providing information on whether the formula triggers an error trace which is dependent on the formula. This interface is shown in figure 7.3, and has two methods. For regular safety properties, it is sufficient to check the state predicates in a single state. For the more complex CTL queries described in this thesis, the sweep-line method provides all predicates in a strongly connected component and the implementing interface must check these against the property under evaluation. When performing a verification of safety properties, the algorithm will call the triggersTrace method for every state predicate in the state space. If the formula to be verified is EF p, and triggersTrace is called with a state that has state predicate p, then the method will return true. When verifying a formula using terminal strongly connected components, the core library is responsible for calling triggersTrace with every terminal strongly connected component. If the formula to be verified is AG EF p, the interface is responsible for knowing that a trace should be triggered if a single state in the terminal strongly connected components satisfy p, and return true if this is the case.

We have chosen the naming "triggers trace" in our implementation with the meaning that a trace is presented when it gives useful information, which is not

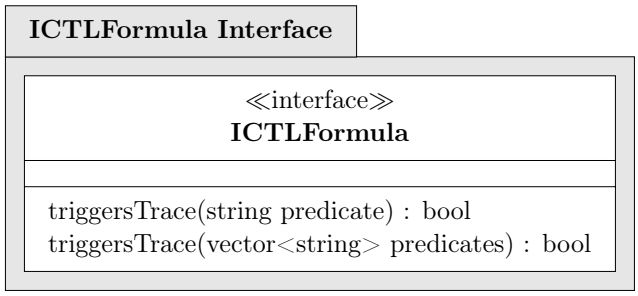


Figure 7.3: Interface for check of CTL properties.

necessarily when a property is violated. For instance, if the desired behaviour of a model is that one should have the possibility to reach a state where the predicate Φ holds, and this property is satisfied, then we present a trace demonstrating the path to this state. If this property is not satisfied because none of the states in the model satisfies Φ, it would not be useful to provide a path to every state in the model as an error trace.

The last important functionality of the sweep-line method is to provide an error trace. The error trace to be provided when a property is triggered, and what information in the state space exploration that needs to be stored, depends upon the property. Figure 7.4 depicts two interfaces for providing error traces; one for automata-based property checking, and one for CTL property checking. They differ in that their respective properties require different error traces according to the property, but they also differ in that the CTL interface only regards the state identity of the states in the model, whereas the automaton interface must also consider the parallel composition of a model and an automaton. In our implementation, we have an object *Node* representing such a composition. The automaton part of a composition is only necessary to provide a correct error trace, and is not part of the error trace itself. The error trace provided therefore only reports the state identities of the model component in the implementation of both interfaces.

We have also defined interfaces where we have extracted all memory management and the use of different containers, e.g., vector, priority queue, map, etc. We have one memory management interface for automata-based property checking and one for CTL property checking. Since we often consider large state spaces in property checking, different implementations of these interfaces can have high impact on the verification time.

For performing the actual property checking, we have defined an interface for each of the classes of properties discussed in this thesis, i.e., one for automata-based property checking and one for CTL property checking. The interfaces are following the *strategy pattern* which is described in section 7.2, and is depicted in figure 7.5. We have two implementations of the automata-based strategy. One

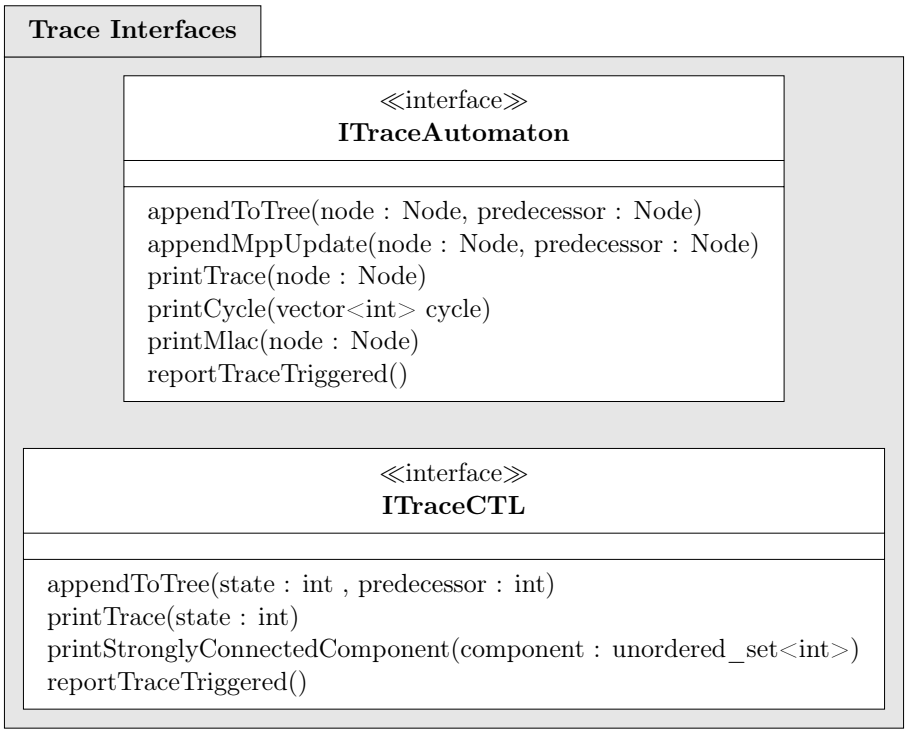


Figure 7.4: Interfaces for providing error traces.

implementation for safety property checking algorithm which was discussed in section 3.3, and one implementation of the LTL algorithm discussed in section 3.4. For CTL property checking we have two implementations of the strategy. One for safety property checking discussed in section 4.2, and one implementation of the algorithm exploiting terminal strongly connected components discussed in section 5.3.

In figure 7.6 and figure 7.7 the dependencies of the two strategies are depicted. The *main* method that is run at start-up has knowledge of both strategies. According to the flags given by a user when the program was executed, the main method is responsible for calling the correct strategy with the appropriate implementation of the strategy. Choosing the right implementation for the respective property checking is done automatically at run-time with the strategy pattern as discussed in section 7.2.

7.2 Strategy Pattern

From [25, p. 315-323] the strategy pattern is used to define a family of algorithms, encapsulate each one, and make them interchangeable. This means that one can use different *strategies* in order to obtain the desired functionality. When the strategy

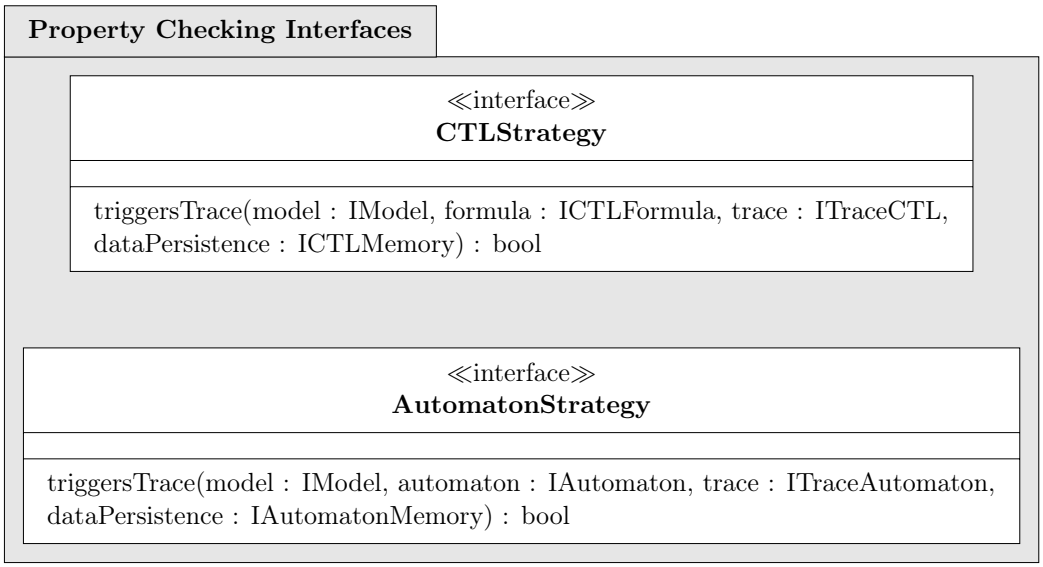


Figure 7.5: Interface for property checking.

pattern is applied, one defines an interface describing the intended behaviour. The context class that uses this behaviour does only know of that interface. All concrete classes that implements this interface can be used interchangeably. This can be a good approach, when for example, one implements two concrete classes where the first focuses on execution time and the second focuses on minimizing the memory usage. Which implementation that is used can change according to the resources requirements in different environments. Another important feature of the strategy pattern is that the concrete strategy can be changed at runtime since the context class only depends on the interface.

When abstracting all behaviour, and using the strategy pattern in our implementation, we support changing the concrete classes. A main reason for the choice of the strategy pattern is the ability to change to the appropriate algorithm depending on the property to be verified at runtime. This is due to the fact that the verification of different properties relies on different state space traversals, and has different requirements for triggering an error trace. The strategy pattern supports this feature in a clean way that also expresses the intent of our source code and highlights the different algorithms that is needed for the sweep-line method.

7.3 Source Code

The choice of using the programming language C++ is mainly done for efficiency. C++ also allows more direct control of memory management to languages that has

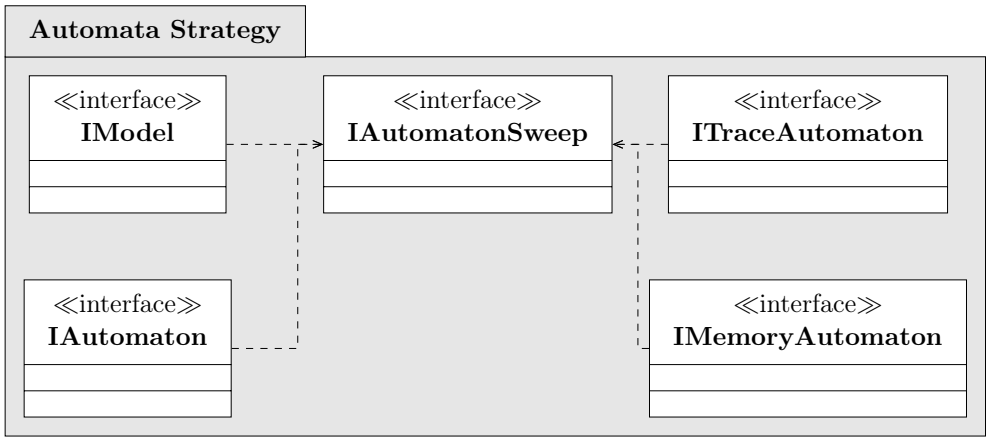


Figure 7.6: Dependencies of the automaton strategy.

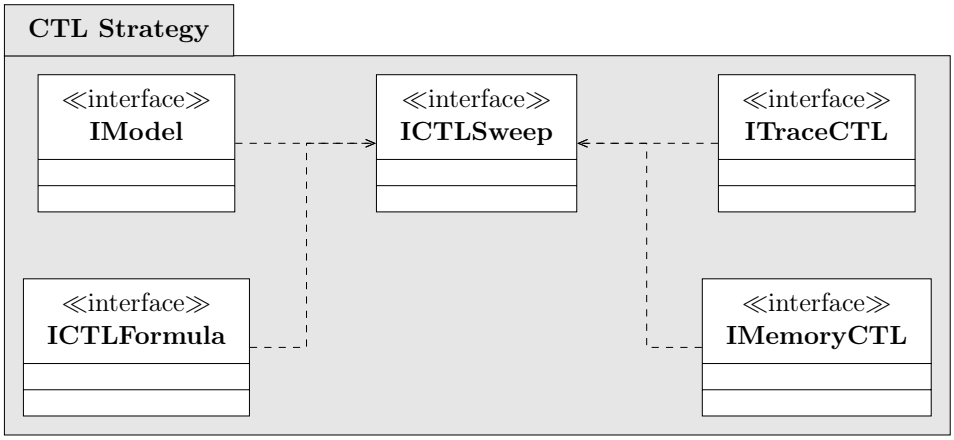


Figure 7.7: Dependencies of the CTL strategy.

automatic memory handling, as for instance Java. We value the efficiency since we often consider large state spaces in explicit-state model checking. Other model checkers are also implemented in C/C++, for instance Spin [5] and Lola [6].

The source code developed for this master's thesis can be found and downloaded from the public Bitbucket repository at [26]. The project consists of 40 files and approximately 1900 lines of code. All source code in this repository is the work of the author except for a getopt-library [27] that is used for reading the command line arguments. The source code was developed with the Visual Studio integrated development environment and readers are therefore encouraged to view the source code in Visual Studio. A free community edition can be downloaded at [28]. With the installation of Visual Studio follows a compiler so that the software can be run directly. We have focused on not to be dependent on the Windows environment and have therefore not used any environment dependent features in our implementation, so the source code can also be compiled with the GNU gcc

compiler [29] and executed in a Linux environment.

The implementation can be compiled with C++ 14. Some features that we want to note in our implementation is the use of smart pointers to avoid memory leaks, the use of auto keyword for deducing variable types, and lambda expressions which allows cleaner code. In listing 7.1, we see three different ways to declare a pointer to a sweep strategy. The benefit of using a smart pointer is that it will keep track of the scope of the pointer and call the delete method on the object that the pointer points to when it goes out of scope. This prevents memory leaks and the programmer is not responsible for the life cycle of the object. The auto keyword is used to deduce the type of the object which lets us write less cluttered code, and we also use the *make_ unique* method which returns a smart pointer.

Listing 7.1: Smart pointer with deduces type.

```
1  // standard pointer to a sweepStrategy
2 ISweepAutomatonStrategy* sweepStrategy = new AutomatonSafety();
3
4  // smart pointer
5 unique_ptr<ISweepAutomaton> sweepStrategy(new AutomatonSafety)
6
7 // smart pointer with deduced type
8 auto sweepStrategy = make_unique<AutomatonSafety>()
```

All containers managing the storage used by the sweep-line method is extracted to an interface. This is done to highlight which kinds of storage operations that are used. The sweep-line method relies upon a progress measure for each state to determine the traversal order. This implies that the priority queue responsible for unprocessed states needs knowledge of the progress values of each state. It is the model interface that holds this information which implies that the priority queue needs knowledge of the model. This is solved with a lambda expression for the custom comparator of the priority queue in listing 7.2.

Listing 7.2: Priority queue with lambda comparator.

```
1 //Declaration in header file
2 unique_ptr<priority_queue<int, vector<int>, function<bool(int, int)>>>
      unprocessed;
3
4 //Constructor in source file
5 function<bool(int, int)> cmp = [model](int state1, int state2) {return
      model->getProgressValue(state1) > model->getProgressValue(state2); };
6
7 unprocessed = make_unique<priority_queue<int, vector<int>,
      decltype(cmp)>>(cmp);
```

We declare in the header file that we have a priority queue that takes a function with two parameters of type int and returns a boolean. In the source file, we see that we create this function with a lambda expression that captures a pointer to the model and returns which of the states that should be processed first according to the progress value of each state. We then create the priority queue *unprocessed* with this compare method. The keyword *decltype* deduces the type of the cmp function and thus we do not need to write *function<bool(int,int)>* one more time.

7.4 Using the Model Checker

Below we explain how one can use the current implementation of the different interfaces.

7.4.1 Model

The current implementation of the IModel interface reads a state space from a file location specified by the user. The implementation asserts a file with one line for each state with the following structure:

- *stateId* The identification of the state.
- *progressValue* The progress value of the state.
- *numberOfPropositions* The number of atomic propositions that is specified in the state.
- List *atomic proposition* List of the truth values of the atomic propositions on the state.
- An alternating sequence of *action successor* Sequence specifying the successor after performing a given action.

The first entry in the file is assumed to be the initial state. Listing 8.2 in section 8.1 is a demonstration of how a state space in file is specified that can be read by our current implementation of the IModel interface.

7.4.2 Automaton

The current implementation of the IAutomaton interface reads an automaton from a file location specified by the user. The implementation asserts a file where the first line gives information of the initial state of the automaton, and the next

line the acceptance states. Thereafter the implementation asserts one line for each state with the following structure:

- *stateId* The identification of the state.
- An alternating sequence of *action successor* Sequence specifying the successor after performing a given action.

The action *true* is recognized as a wild card meaning that this action is always enabled in the automaton. Listing 7.3 is a demonstration of how the automaton in figure 8.2b in section 8.2 is stored in a file.

Listing 7.3: Automaton from file.

```
0
1
0 true 0 p!q 1
1 !q 1
```

7.4.3 Formula

The current implementation of the ICTLFormula interface reads a formula from a file location specified by the user. The implementation asserts a file with one line specifying the formula. The formulas that can be recognized are:

- EF Φ
- AG Φ
- EFAG Φ
- AGEF Φ

where Φ can be a single string representing the state predicate.

7.4.4 Trace

The implementation of the trace interface for both automaton-based property checking and computation tree logic property checking has hard-coded the file names that they use to store information and provide a trace. They will store the tree in a file called *tree.txt*. The mpp update information is stored in *mppUpdate.txt*.

On large state spaces, these files may not fit to internal memory. To provide a trace one must search these files in external memory for the predecessor relations.

The resulting trace is written to a file called *trace.txt* in the same folder as the tool is executed.

7.4.5 Using the interfaces

An important benefit gained through the definition the interfaces discussed in section 7.1 is that the core of the sweep-line method implemented in this thesis rely only upon the methods defined by the interfaces. This means that one can easily provide own implementations of these interfaces and still utilize the algorithms implemented in this thesis.

7.5 Compiling and Executing

Compiling the source code can be done in different ways. The code can be compiled and executed in Visual Studio. Listing 7.4 demonstrates how the source code can be compiled and executed from a command line with both a Windows command line tool or a terminal in Linux.

Listing 7.4: Compiling source code from command line.

```
1 //Windows C++ compiler
2 cl /EHa /FeSweepline.exe /std:c++14 *.cpp
3
4 //GNU gcc compiler
5 g++ -std=c++14 *.cpp -o Sweepline.out
```

Both commands will compile the source code and generate an executable named Sweepline. Compilation is done according to the C++ 14 standard without any compiler optimization flags.

Executing the compiled source without any flags one will be presented with a help message for the tool shown in listing 7.5.

Listing 7.5: Help message of the Sweepline tool.

```
1 Usage: sweepline
2 Options:
3 -m      File path to model
4
5 -s      Check for safety properties
6 -l      Check for LTL properties
7 -c      Check for CTL properties
8
```

```
-f      File path to CTL formula
-a      File path to automaton

-h      Show this help message
```

Our tool needs several flags specified when executed. The -m flag takes the file path to the system model and is required. Next we must specify one of the properties that we want to check with the -s, -l or -c flag, and depending on whether we want to check a computation tree logic (CTL) property or an automata-based property, we must specify the file path to either the CTL formula or the file path to the automaton. Using our tool to check a safety property in the model *testModel.txt* for the CTL predicate which is located in the file *testPredicate.txt* we can execute the tool as shown in listing 7.6.

Listing 7.6: Execution of the Sweepline tool.

```
Sweepline -m testModel.txt -f testPredicate.txt -s
```

The sweepline tool will then go through the state space and perform the verification of the property. In case an error trace is triggered the tools report that this is the case, and writes the error trace to a text file called trace.txt in the same folder as the tool was executed.

Chapter 8

Experimental Validation

In this chapter, the implementation of the sweep-line method is validated using an example in the form of a stop-and-wait protocol. We briefly introduce the stop-and-wait protocol that is used as model, and explain how we specify the properties that are to be verified. We validate safety properties for both automata-based property checking and computation tree logic (CTL) properties. We also validate linear temporal logic (LTL) properties and CTL properties with monotonic progress measures.

8.1 Stop-and-Wait Protocol

To validate our implementation, a stop-and-wait protocol is used. The protocol is a method for communicating over a network. The *sender* transmits one *packet* at a time, and waits for an acknowledgement from the *receiver* for that packet before transmission of the next packet. If no acknowledgement is received within a certain amount of time, the sender retransmits the packet. The retransmission of packets that are unacknowledged ensures that information is not lost if the network is unreliable, i.e., if packet loss can occur.

The protocol is depicted in figure 8.1 and is modelled using Coloured Petri Nets and CPN Tools [30]. In figure 8.1 the has two transitions: *Send Packet*, and *Receive Ack* models a sender. The sender transmits packets to a receiver. The receiver is modelled with the transition *Receive Packet*. The receiver transmits acknowledgement when packets are received. The places between *Transmit Packet* - *Receive Packet* and *Transmit Ack* - *Receive Ack* are the network that the protocol operates on. We have defined several parameters determining how the protocol behaves. This makes it possible to specify different properties that the protocol possesses, in order to validated the sweep-line implementation. The first parameter

is the *Succ* function located at the two transmit transitions, *Transmit Packet* and *Transmit Ack*. With this parameter, we can decide whether there is possibility for packet loss. The next parameter is the *Ack* function at the transition *Receive Packet* where we decide whether all packets received are acknowledged or just the packet the receiver is currently waiting on. The last parameter is the *domax* function on the transition *Receive Ack* where we decide which packet to send next when an acknowledgement is received. If the receiver acknowledges all packets, and we do not take the maximum of the previously received acknowledgement, and the currently received acknowledgement, then we can have the possibility that an old acknowledgement is chosen. This would lead to that the protocol will resend a packet that already has been acknowledged, thus making regress.

The size of the state space is determined by the number of packets to be sent and the *limit* of how many packets that can be on the network at the same time. The progress measure is defined to be a function of which packet is the next to be sent, and which is the next to be received, where the packet to be sent is the most significant factor. In listing 8.1 we see the progress measure function written in the functional programming language ML.

Listing 8.1: Progress measure function for the stop-and-wait protocol.

```
1 fun SWProgressMeasure n =
2 IntInf.fromInt(
3       (hentPakkerParam()) * ms_to_col(Mark.Protocol'NextSend 1 n) +
4       ms_to_col(Mark.Protocol'NextRec 1 n));
```

From CPN Tools one can export the resulting state space from the model. In addition to exporting the state space one can define atomic propositions on each state. The values and meaning of the atomic propositions will vary in each test case and are defined with respect to the places of the model and the property we want to check. When conducting model checking with the sweep-line method the properties are specified over these atomic propositions. From CPN Tools one can compute the complete state space and dump it to an external file which we parse before conducting the property checking using our implementation.

In listing 8.2 are the ten first entries of a state space dump from CPN Tools. Each state has two atomic propositions specified over the alphabet of p and q. For the first entry, we see that this is state 1 which has a progress value of 7. It has two atomic propositions where neither p nor q holds. State 1 has the single successor 2, and * is the action performed when going from state 1 to state 2. The * action is *internal*, meaning that the model does not give us specific information of which action performed. We see that in the successor list of state 100, the action RP2 is performed when going to state 133, which stands for Receive Packet 2. Our command line tool is only considering the atomic propositions and does not consider the actions on the edges.

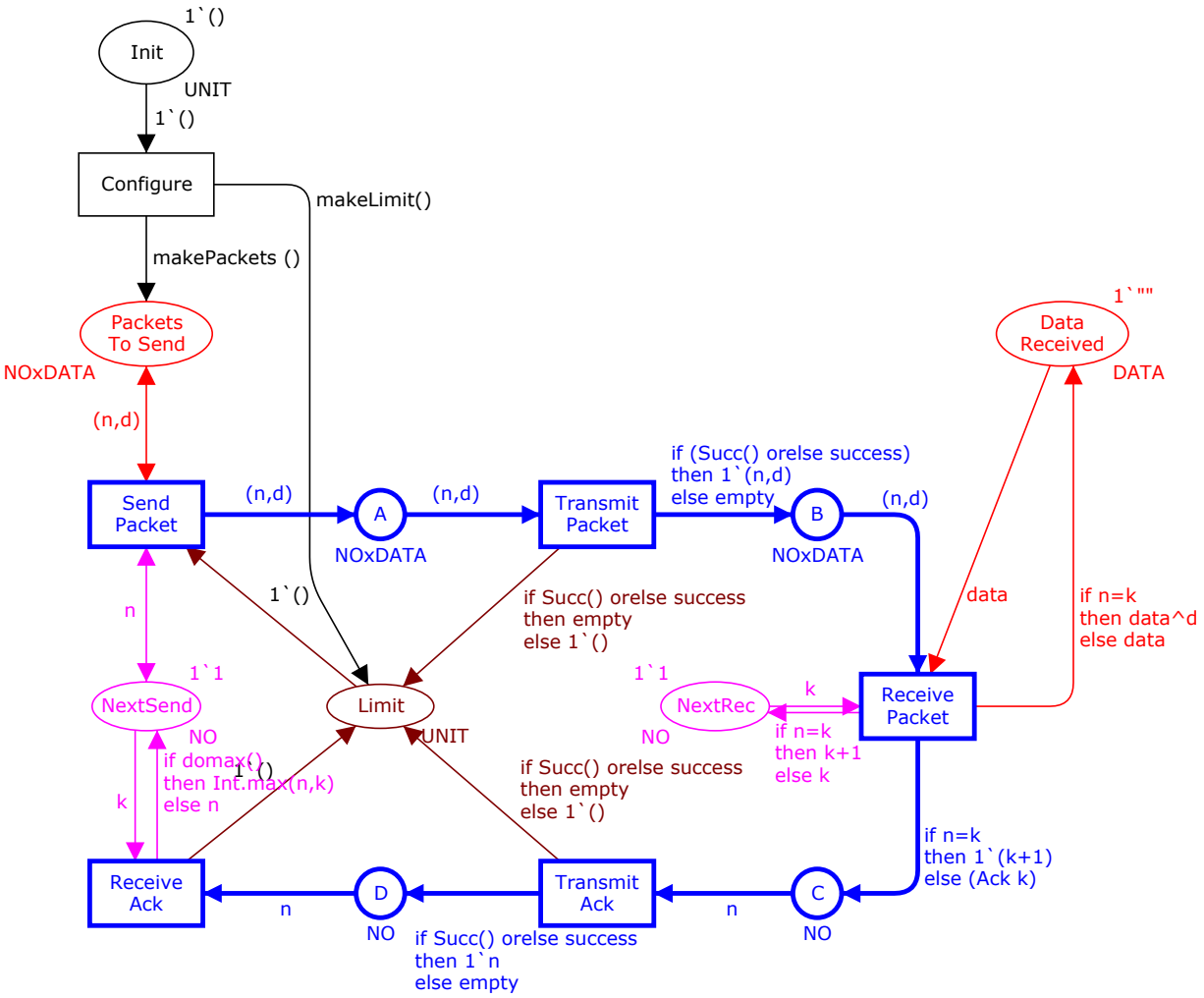


Figure 8.1: Stop-and-wait protocol in CPNTools.

Listing 8.2: State space dumped from CPN Tools.

```
1 1 7 2 !p !q * 2
2 10 8 2 !p !q * 16
3 100 15 2 !p q * 134 RP2 133 * 131
4 1000 28 2 !p !q * 1206 * 533 * 1200
5 10000 25 2 !p !q * 9137 * 10718 * 9204 * 10725
6 10001 25 2 !p !q * 9138 * 10719 * 7480 * 10724
7 10002 24 2 !p !q * 9140 * 10720 RP6 10726 * 3154
8 10003 25 2 !p !q * 9141 * 10721 * 10728 * 10727 * 7480 * 10726
9 10004 25 2 !p !q * 10729 * 9150 * 10722 RP6 7480
10 10005 25 2 !p !q * 9151 * 10723 * 9204 * 10729 * 9208 * 10730
```

For the validation, only state spaces that can be stored in internal memory are considered. We have an implementation of the IModel interface that reads the file containing the state space and parses it to be able to satisfy the public methods defined by the interface.

8.2 Test Cases

We have defined several test cases in order to validate our sweep-line implementation. The different test cases are obtained by different configurations in at the CPN model, and truth values of the atomic propositions are exported with the state space according to query functions in CPN Tools. The properties we want to validate are:

Safety Property: All packets are received

For this property, we have a predicate p on each state indicating if all packets are received, and we thus needs to search the state space for a state where p is true. We perform the verification of the following properties:

- Automata-based with the safety automaton in figure 8.2a.
- CTL with the formula EF p.

In listing 8.3 we see the function that is used to determine the truth value for each state.

Listing 8.3: Function determining if all packets are received.

```
1 fun SWAllReceivedPredicate n =
2     let
3      val all_received = String.concat (List.map (fn (_,p) => p)
           (makePackets (ms_to_col (Mark.Protocol'Init 1 1))))
4      val received = ms_to_col (Mark.Protocol'Data_Received 1 n)
5 in
```

```
6   all_received = received
7 end;
```

Safety Property: Sender is one sequence number higher than the receiver

For this property, we have a predicate p on each state indicating if this is the case and we need to search the state space for a state where p is true. We perform the verifications of the following properties:

- Automata-based model checking with the safety automaton in figure 8.2a.
- CTL model checking with the formula EF p.

In listing 8.4 we see the function that is used to determine the truth value for each state.

Listing 8.4: Function determining if sender is one sequence number higher than receiver.

```
1 fun TwoAhead n =
2   let
3       val sendseq = ms_to_col (Mark.Protocol'NextSend 1 n)
4       val recvseq = ms_to_col (Mark.Protocol'NextReceive 1 n)
5   in
6       sendseq - 2 >= recvseq
7   end;
```

Branching Property: All paths exist finally (all packets received).

For this property, we have a predicate p on each state indicating if all packets have been received. We perform the verification of the following property:

- CTL model checking with the formula AG EF p

We use the function from listing 8.3 to determine the truth values of each state.

Linear Time Property: Always (packet sent $\Longrightarrow$ Eventually acknowledgement will be sent)

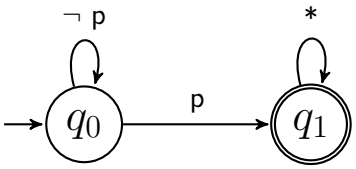
For this property, we have two predicates p and q indicating packet sent and acknowledgement sent respectively. We construct the automaton of the negated LTL formula from the property specified and perform the verification of the following property:

- Automata-based model checking with the automaton in figure 8.2b

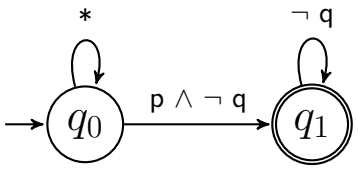
In listing 8.5 we see the function that is used to the truth values for each state.

Listing 8.5: Function determining if packet sent is acknowledged.

```
1 fun SendPacketPred i n = List.exists (fn (j,_) => j = i)
    (ms_to_list (Mark.Protocol'A 1 n));
2 fun TransmitAckPred i n = List.exists (fn j => j = i + 1)
    (ms_to_list (Mark.Protocol'D 1 n));
```



(a) Safety automaton.



(b) Negated automaton of LTL formula $\Box(p \Rightarrow \Diamond\ q)$.

Figure 8.2: Automata used in validation.

We use these properties to define several test cases. We want to check the properties in both monotonic progress measures and non-monotonic progress measures. We also alter which properties the model possesses by specifying the possibility for packet loss and which acknowledgement packets that are sent. In chapter 8.3, we perform each test case, specify which properties the model possesses and validate the results in terms of the expected results.

8.3 Validation

All state spaces used in the validation can be found in the validation folder in the repository containing the source code [26]. We compiled the source code with the G++ compiler to an executable named *sweepline*, with the optimization flag O2. For each test case, we will list the parameters that the test was run with. This ensures that the validation tests can be easily reproduced. From the root folder in the repository, the source code is in the /Sweepline folder and all commands given in this chapter asserts that they are executed from this folder.

- **Compilator:** GNU Compiler Collection G++ 5.4.0
- **Compilation:** g++ -std=c++14 -O2 *.cpp -o sweepline
- **Git commit identification:** f37a4b9644802a47e247a37b425ef5ceb6c18429
- **Git branch:** master branch

We will go through the different validation tests that were specified in chapter 8.2 and specify which properties that are expected by the state space. The automata

used in the verifications are in the /Automata folder, and the CTL formulas are in the /CTLFormulas folder.

Safety property: All packets are received
We have computed two state spaces satisfying this property. The state space in file *swprotocol-6-3-EFallreceived-true.ss* is computed from a model with a monotonic progress measure, and the state space in file *swprotocol-6-3-EFallReceived-true-nm.ss* is computed with a model with a non-monotonic progress measure. For both state spaces, we expect the property to be satisfied and a trace provided from the initial state to a state where predicate p is true. Since this is a safety property, it can be checked automata-based or with CTL. We want to use both and listing 8.6 shows the four executions testing both state spaces with the two ways to express the properties.

Listing 8.6: Validating safety properties that is satisfied.

```
1 //Computation tree logic checking
2 sweepline -m validation/swprotocol-6-3-EFallreceived-true.ss -f
    CTLFormulas/EF-p.txt -s
3 sweepline -m validation/swprotocol-6-3-EFallreceived-true-nm.ss -f
    CTLFormulas/EF-p.txt -s
4
5 //Automata-based checking
6 sweepline -m validation/swprotocol-6-3-EFallreceived-true.ss -a
    Automata/pPred.txt -s
7 sweepline -m validation/swprotocol-6-3-EFallreceived-true-nm.ss -a
    Automata/pPred.txt -s
```

All executions report that a trace is triggered, which is what we expected.

Safety Property: Sender is one sequence number higher than the receiver
We have computed two state spaces satisfying this property. The state space in file *swprotocol-6-3-EFtwoahead-false.ss* is monotonic, and the state space in file *swprotocol-6-3-EFtwoahead-false-nm.ss* is non-monotonic. This property is not satisfied and no states in these state spaces will satisfy this property. We will check this property with automata-based checking and with CTL checking. Listing 8.7 shows the executions.

Listing 8.7: Validating safety properties that are not satisfied.

```
1 //Computation tree logic checking
2 sweepline -m validation/swprotocol-6-3-EFtwoahead-false.ss -f
    CTLFormulas/EF-p.txt -s
3 sweepline -m validation/swprotocol-6-3-EFtwoahead-false-nm.ss -f
    CTLFormulas/EF-p.txt -s
```

```
4
5 //Automata-based checking
6 sweepline -m validation/swprotocol-6-3-EFtwoahead-false.ss -a
    Automata/pPred.txt -s
7 sweepline -m validation/swprotocol-6-3-EFtwoahead-false-nm.ss -a
    Automata/pPred.txt -s
```

None of the executions reports that a trace is triggered, which is what we expected.

Branching Property: All paths exist finally (all packets received)
We have computed two state spaces for this property. The state space *swprotocol-6-3-AGEFallreceived-true.ss* satisfies the property, whereas the state space *swprotocol-6-3-AGEFallreceived-false.ss* does not. The algorithm for checking these properties can only be used for monotonic progress measures, so this is a constraint for both state spaces. This is a pure branching time property and can only be checked with CTL. Listing 8.8 shows the executions.

Listing 8.8: Validating CTL properties.

```
1 sweepline -m validation/swprotocol-6-3-AGEFallreceived-true.ss -f
    CTLFormulas/AGEF-p.txt -c
2 sweepline -m validation/swprotocol-6-3-AGEFallreceived-false.ss -f
    CTLFormulas/AGEF-p.txt -c
```

The state space *swprotocol-6-3-AGEFallreceived-true.ss* satisfies the property, meaning that all terminal strongly connected component contains a state that satisfy the predicate. For this execution no trace was reported, which is expected.

For the state space *swprotocol-6-3-AGEFallreceived-false.ss* a trace was reported with a terminal strongly connected component where no state satisfied the predicate, which was expected.

Linear Time Property: Always (packet sent $\Longrightarrow$ Eventually acknowledgement will be sent)
We have computed four state space for validating this property. We have two state spaces satisfying the property, where one is monotonic and one is non-monotonic. We also have two state spaces not satisfying the property, where one is monotonic, and one is non-monotonic. This property will be checked with automata-based property checking. Listing 8.9 shows the executions.

Listing 8.9: Validating LTL properties.

```
1 sweepline -m validation/swprotocol-6-3-AsendimpEtransmit-false.ss -a
    Automata/ApImpEqComplement.txt -l
```

```
2 sweepline -m validation/swprotocol-6-3-AsendimpEtransmit-false-nm.ss -a
    Automata/ApImpEqComplement.txt -l
3
4 sweepline -m validation/swprotocol-6-3-AsendimpEtransmit-true.ss -a
    Automata/ApImpEqComplement.txt -l
5 sweepline -m validation/swprotocol-6-3-AsendimpEtransmit-true-nm.ss -a
    Automata/ApImpEqComplement.txt -l
```

The state space *swprotocol-6-3-AsendimpEtransmit-false.ss* does not satisfy the property, and as expected we were presented with an error trace. The state space *swprotocol-6-3-AsendimpEtransmit-true.ss* satisfies the property and therefore no trace was reported.

When conducting the same test on the model with non-monotonic progress measures, the algorithm terminates when a single layer acceptance cycle has been found. Since the algorithm for finding multi layer acceptance cycle is executed with the set of persistent states we must perform a full state space traversal before the MLAC algorithm is executed. By commenting out the source code reporting SLACs we are able to execute the MLAC algorithm with the set of persistent states. However, the algorithm only reports SLACs. In order to complete the search, we have prevented the algorithm from terminating when a cycle is reported, leading it to report all cycle in the state space.

By inspection the cycles, we found that none of the cycles reported are MLACs. Therefore, we are not sure if the state space possesses cycle that span over multiple layers. We are, however, satisfied with that the algorithm is able to detect acceptance cycles, and that traces for these are reported.

All state spaces used in our validation tests have less than 15 000 states and are considered small in the context of model checking. All executions terminate within seconds, except for the test when we search and report all cycles in *swprotocol-6-3-AsendimpEtransmit-false-nm.ss*. This test takes approximately twenty minutes, due to I/O performance in reporting all cycles.

In table 8.1, we summarize the validation tests done in this section. We have omitted the prefixes of the models, i.e., EFallreceived-true.ss instead of *swprotocol-6-3-EFallreceived-true.ss*. The folder name of the automaton and CTL formula is also omitted.

Table 8.1: Summary of validation tests.

		Trace	
Model	**CTL Formula**	**Expected**	**Triggered**
EFallreceived-true.ss	EF-p.txt	yes	yes
EFallreceived-true-nm.ss	EF-p.txt	yes	yes
EFtwoahead-false.ss	EF-p.txt	no	no
EFtwoahead-false-nm.ss	EF-p.txt	no	no
AGEFallreceived-false.ss	AGEF-p.txt	yes	yes
AGEFallreceived-true.ss	AGEF-p.txt	no	no
	Automaton		
EFallreceived-true.ss	pPred.txt	yes	yes
EFallreceived-true-nm.ss	pPred.txt	yes	yes
EFtwoahead-false.ss	pPred.txt	no	no
EFtwoahead-false-nm.ss	pPred.txt	no	no
AsendimpEtransmit-false.ss	ApImpEqComplement.txt	yes	yes
AsendimpEtransmit-false-nm.ss	ApImpEqComplement.txt	yes	yes
AsendimpEtransmit-true.ss	ApImpEqComplement.txt	no	no
AsendimpEtransmit-true-nm.ss	ApImpEqComplement.txt	no	no

Chapter 9

Conclusion and Future Work

In this chapter, the work done in this thesis is summed up and linked to the research questions forming the basis of our thesis. We provide a discussion of the results before we suggest areas of further work.

9.1 Contributions

In this thesis, we have focused on explicit-state model checking with the sweep-line method. Since the sweep-line method deletes states on-the-fly, the standard algorithms for property checking are not directly compatible with the sweep-line method. Therefore tailored algorithms for performing property checking is needed. Most of the research in this thesis has been on how to support property checking with the sweep-line method, but we have also focused on implementing a console application with the algorithms for the sweep-line method. The main contributions in this thesis are:

- A new general algorithm for providing error traces for linear temporal logic (LTL) property checking in combination with the sweep-line method.
- A new algorithm for checking important computation tree logic (CTL) properties with the sweep-line method.
- Defined interfaces that the sweep-line method depends upon for conducting a state space traversal.
- Corrected an error in the pseudo code in [4] that lead to higher peak memory usage and premature termination.
- Implementation of the algorithms presented in this thesis for the sweep-line method that results in a command line tool for performing property

checking.

From chapter 1.7, we recall that the research questions of this thesis are:

1. Property checking with the sweep-line method compared to other state space traversal methods such as DFS or BFS
   (a) What is the main difference with the sweep-line method compared to the standard techniques?
   (b) How to support the verification of different sets of properties?
   (c) How to provide error traces?
2. Unified implementation
   (a) How to implement the different variations in a unified way?
   (b) Which interfaces must the sweep-line method depend on for the representation of a model, property to be verified and memory management?

Research question 1a is answered in chapter 2 where we have introduced the sweep-line method. We explained how it performs on-the-fly state space traversal, divide the state space into layers and how we must recognize persistent states in order to guarantee termination. We have in chapter 3 and chapter 4 described the standard way of doing property checking without the sweep-line method. For research question 1b we described in chapter 3 how safety properties and LTL properties can be checked with the sweep-line method and how parallel composition can be done on the fly with the sweep-line method. In chapter 4, we described how CTL safety properties can be verified, before we in chapter 5 developed a new algorithm that exploits strongly connected components in state spaces with monotonic progress measures to check the CTL properties AG EF Φ and EF AG Φ. Research question 1c is regarding how to provide error traces with the sweep-line method. We have in chapter 6 presented the method for providing error trace for safety properties. We developed a new general algorithm for proving error traces for LTL property checking with the sweep-line method, and we developed a new algorithm for providing error traces for the CTL properties in chapter 5.

The research questions about unified implementation are answered in chapter 7. For question 2a, we have described how the strategy pattern can be used to support a unified implementation in a clean way. This pattern allows us to run the correct algorithm at run time depending upon the property to be verified. For question 2b, we have presented the interfaces required for the sweep-line method for performing a verification. By developing the command line tool, we have also demonstrated that the implementation of the algorithms discussed in this thesis works. The repository with the source code and validation examples used in this

thesis are publicly available, and thus our experiments are easily reproducible. The qualitative results in this thesis regarding the unified implementation and strategy pattern can be examined by inspecting the source code.

9.2 Discussion

We have provided proof of correctness for the general algorithm for providing LTL error traces. We have also demonstrated with validation examples that the implementation works. We do however find that the implementation should be tested thoroughly on both small examples trying to find corner cases and several large examples for industrial sized models before concluding correctness of the implementation. A feature is that the algorithm is easily implemented and that we maintain the time complexity of the original LTL algorithm for the sweep-line method. In addition to the file on external storage for providing the path from the initial state to the acceptance cycle, we only need one more file on external storage for providing the LTL error trace.

For the new CTL algorithms developed in this thesis it is important to highlight that we were able to integrate Tarjan's algorithm for strongly connected components with the on-the-fly sweep-line state space traversal. We thereby avoid that the sweep-line method computes one layer at a time, before running Tarjan's algorithm after each layer has been computed. We also maintain the time complexity of Tarjan's algorithm when integrating it with the sweep-line method which is a desired property.

For checking CTL properties with the sweep-line method we clearly have some limitations in the algorithm developed in this thesis. First of all, we can only perform property checking on state spaces with monotonic progress measures which can have several drawbacks in verification time and memory requirements. Furthermore, a monotonic progress measure may not be a feasible constraint on some models. We are only able to check two forms of properties and have not been able to generalize them or otherwise discuss the possibilities for checking additional properties in CTL with the sweep-line method.

The command line tool developed in this thesis can be used to verify properties with the sweep-line method and we have validated its correctness with the stop-and-wait protocol and a suite of different properties. In order to be more robust there should have been written unit tests and executed several validations to find and correct potential implementation bugs. The state spaces used to validate the tool have only been state spaces that we could store in internal memory. We argue that this is good enough to demonstrate the algorithms implemented in this thesis, but there may be small implementation details that must be changed

when running the tool on larger state spaces. To do so, one must implement the model interface in such a way that successors are computed on the fly, whereas our current implementation reads the whole state space from an external file at start up.

The error traces provided when a property is violated contains just the state identification of each state in the trace. With only this information it can be difficult to find and correct the model, especially if the error trace is long. To provide an error trace with more information or the possibility to connect the error trace with the model would be useful when trying to understand and correct an error in a system model.

Overall we are satisfied with the findings in this thesis and the contributions made. We conclude that we have answered the research questions which formed the basis of the master's thesis.

9.3 Future Work

In this master's thesis, we have developed a new algorithm for checking two key CTL properties when considering monotonic progress measures. As we have discussed, we have not formalized or provided a generalization of the properties we verify. It would require future work to investigate if there are more properties that can be verified with strongly connected components or other features that state spaces with monotonic progress measures possess. It would also be interesting to investigate if one could check properties without enforcing the state spaces to be monotonic. Since we do not delete the persistent states in non-monotonic state spaces, we have the possibility to store information in them, or otherwise exploit the persistent states as with the linear temporal logic algorithm for the sweep-line method.

Another interesting topic for future research is how to provide a progress measures for a state space. Since the progress measure can be user-defined and is not necessarily directly obtained from the model itself one can experiment with different progress measures. It would also be interesting to investigate and formalize a definition of what it means for a progress measure to be "optimal". For a monotonic progress measure one could argue that the optimal progress measure is defined according to minimize peak memory. For non-monotonic progress measures we have regress edges leading to persistent states that may need to be explored several times. The states stored in memory at any time are the states in the current layer, and all states added as unprocessed which resides in future layers. A progress measure with "small" layers could mean that peak memory is still high as many states are added to the queue of unprocessed too be explored

in future layers. Minimizing the peak memory could on the other hand mean that there exist several persistent states and thus yielding very long verification times caused by re-exploration for every persistent state. One possibility would be to set a threshold for memory requirements and compute a progress measure with fewest persistent states satisfying the memory requirements. We have only considered defining a progress measure for the system model in this thesis, but for automata-based model checking one could define a progress measure on the automaton and then define a new progress measure for the product of the two in the parallel composition. That may further reduce peak memory usage.

For the command line tool developed with this thesis, relevant future work would be to develop different implementation of the memory management interface and conduct experiments on large state spaces to see if there could be performance differences in how it is implemented. Since we are relying upon writing and reading from external storage, investigating how this can be done most efficiently is of importance. A concrete idea is to write to external memory in a separate thread once for each layer right before the deletion of states. Further work on the command line tool could be to make it more robust and give the user more flexibility to specify models, automata and CTL formula but also how the error trace should be provided. In order to be easier to use on a broader set of models an integration to other model checkers like Spin [5] and Lola[6] that has support for modelling protocols is of special interest.

Bibliography

[1] N. G. Leveson and C. S. Turner, "An investigation of the therac-25 accidents," *Computer*, vol. 26, no. 7, pp. 18–41, 1993.

[2] Wikipedia, "Pentium fdiv bug — Wikipedia, the free encyclopedia." `https://en.wikipedia.org/wiki/Pentium_FDIV_bug`, 2016. [Online; accessed 3-March-2016].

[3] C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.

[4] K. Jensen, L. M. Kristensen, and T. Mailund, "The sweep-line state space exploration method," *Theor. Comput. Sci.*, vol. 429, pp. 169–179, 2012.

[5] Spin. `http://spinroot.com/spin/whatispin.html`, 2016. [Online; accessed 3-March-2016].

[6] Lola. `http://service-technology.org/lola/`, 2016. [Online; accessed 4-March-2016].

[7] A. Valmari, "The state explosion problem," in *Lectures on Petri nets I: Basic models*, pp. 429–528, Springer, 1998.

[8] Promela, "Promela manual pages." `http://spinroot.com/spin/Man/promela.html`, 2016. [Online; accessed 4-March-2016].

[9] W. Reisig and G. Rozenberg, *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, vol. 1491 of *Lecture Notes in Computer Science*, pp. 122–173. Springer Berlin Heidelberg, 1998.

[10] M. T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis and Internet Examples*. New York, NY, USA: John Wiley & Sons, Inc., 2nd ed., 2009.

[11] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, pp. 321–, July 1961.

[12] P. Sanders, *Algorithm Engineering – An Attempt at a Definition*, pp. 321–340. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[13] M. Muller-Hannemann and S. Schirra, eds., *Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice*. Berlin, Heidelberg: Springer-Verlag, 2010.

[14] S. Evangelista and L. M. Kristensen, "A sweep-line method for büchi automata-based model checking," *Fundam. Inform.*, vol. 131, no. 1, pp. 27–53, 2014.

[15] S. Evangelista and L. M. Kristensen, "Hybrid on-the-fly LTL model checking with the sweep-line method," in *Application and Theory of Petri Nets - 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25-29, 2012. Proceedings* (S. Haddad and L. Pomello, eds.), vol. 7347 of *Lecture Notes in Computer Science*, pp. 248–267, Springer, 2012.

[16] L. Brim, I. Černá, P. Moravec, and J. Šimša, *Accepting Predecessors Are Better than Back Edges in Distributed LTL Model-Checking*, pp. 352–366. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004.

[17] A. Cheng, S. Christensen, and K. H. Mortensen, "Model checking coloured petri nets exploiting strongly connected components," in *Proceedings of the International Workshop on Discrete Event Systems, WODES96. Institution of Electrical Engineers, Computing and Control Division*, pp. 169–177, 1997.

[18] R. Tarjan, "Depth first search and linear graph algorithms," *SIAM JOURNAL ON COMPUTING*, vol. 1, no. 2, 1972.

[19] V. Bloemen, "On-the-fly parallel decomposition of strongly connected components," Master's thesis, University of Twente, June 2015.

[20] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67 – 72, 1981.

[21] GeeksforGeeks, "Tarjan's algorithm to find strongly connected components." `http://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components`, 2016. [Online; accessed 28-April-2017].

[22] L. M. Kristensen and T. Mailund, "Efficient path finding with the sweep-line method using external storage," in *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings* (J. S. Dong and J. Woodcock, eds.), vol. 2885 of *Lecture Notes in Computer Science*, pp. 319–337, Springer, 2003.

[23] isocpp.org, "The ISO C++ standard." `https://isocpp.org/std/the-standard`, 2014. [Online; accessed 10-May-2017].

[24] cppreference, "cppreference." http://en.cppreference.com/w/, 2017. [Online; accessed 10-May-2017].

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[26] A. Lilleskare, "Model checking with the sweep-line method." https://bitbucket.org/exoen/sweepline, 2017. [Online; accessed 10-May-2017].

[27] M. K. Kim, "getopt.c." ftp://ftp.dante.de/tex-archive/support/tr2latex/getopt.c, 2017. [Online; accessed 18-May-2017].

[28] "Visual Studio community edition." https://www.visualstudio.com/vs/community/, 2017. [Online; accessed 10-May-2017].

[29] "GCC, the GNU compiler collection." https://gcc.gnu.org/, 2017. [Online; accessed 10-May-2017].

[30] "CPN Tools." cpntools.org, 2017. [Online; accessed 18-May-2017].

Glossary

acceptance cycle A cycle that contains at least one state satisfying some predicate.

atomic proposition A property that holds in a state, and which cannot be further subdivided.

maximal persistent predecessor The maximal persistent state s that is backward reachable from some state s'.

nested depth-first search A graph traversal algorithm that finds a cycle containing at least one node that satisfy some predicate.

non-deterministic choice A choice made that cannot be predicted in advance.

parallel composition A parallel (or product) composition is a composition of two graphs e.g. models, automata or transition system, representing their combined behaviour according some synchronisation rules.

persistent state A state that cannot be deleted from memory.

progress measure A measurement which defines a total order on a transition system according to the level of progress the system has made in each state.

regress edge An edge where the successor state has a strictly lower progress measure associated with it than the predecessor according to the progress measure.

state space Directed graph representing all possible behaviours of a system model.

strongly connected component A maximal set of states in a directed graph where each state s, can reach every other state s'.

Acronyms

BFS breadth-first search.

CTL computation tree logic.

DFS depth-first search.

LTL linear temporal logic.

MLAC multi-layer acceptance cycle.

NBA non-deterministic Büchi automaton.

NFA non-deteministic finite automaton.

SLAC single layer acceptance cycle.

TS transition system.

List of Figures

List of Algorithms

Listings

List of Definitions and Theorems