Introduction into Model Checking
An Approach for Parallelization

Seminar Paper

Authors

Gino Paulaitis and Stefan Derungs

Advisor

Prof. Josef Joller

Published at

Department of Computer Science

HSR University of Applied Sciences Rapperswil

Rapperswil, Switzerland

Fall Term 2012

Abstract

This paper deals with *model checking*, a branch of the software verification domain. As of model checking needing a lot of computation and time resources and in view of the big amount of available cores in today's graphic cards our aim was to analyze possible ways for massive parallelization of model checking. This parallelization could eliminate its main drawback, namely the high time costs.

Because of the complexity of the topic and our nonexistent previous knowledge, in the first part an introduction into model checking is given. Its purpose is to provide a basic overview of what model checking is and how it works. Afterwards, we tried to identify several points for parallelization and analyzed some of them further.

In our opinion there exists a certain potential for parallelization. However, the time to finish this seminar paper was limited and further analysis of the potential would require even more familiarization with model checking and thus more time. On the one hand, the problems to solve belong to the most complex ones, i.e. in computational complexity theory you move in the field of NP-hard problems. On the other hand, if you want to parallelize as much elements of model checking as possible, such a parallelized algorithm would need to be implemented partially or even completely from scratch.

Contents

Authors Note

This paper is based on the book "Principles of Model Checking" by Christel Baier and Joost-Pieter Katoen [3]. Hence, many sections are related very closely to this book. For simplicity, there will not be any inline referencing for this book.

Chapter 1

Introduction

This chapter will introduce some software verification techniques with their obvious advantages and drawbacks. It should provide a basic understanding for the following chapters.

1.1 Software Verification[1]

Computers and Software are becoming more and more important in everybody's life. The times where only governmental projects and some niche products depended on them are long gone. Today, pieces of software can be found almost everywhere - a patients drip providing him with vital medicine, destination boards at train stations, the on-board computer in your car keeping you up to date on whatever happens in and around your car, your TV, coffee machine, cellphone or alarm clock... The list is almost endless. However, all these systems have one common denominator: humans. Our lives depend on them. For this reason, the correct functioning of software and electronic devices is crucial. Software verification has therefore been a topic that engineers have been dealing with for a long time.

The software verification domain can be divided in two main areas: dynamic and static verification. Dynamic verification concentrates on testing a specific implementation during its execution. Static verification, by contrast, analyses software specifications and formal definitions without any execution. In this paper we will concentrate on the latter - namely on a technique called *model checking*.

Before we dive further into the topic, we will introduce two terms - system and correctness. A *system* is an entity specified by properties and requirements. A system or software will be considered to behave *correctly* if all the properties and requirements defining this very system are fulfilled.

1.1.1 Verification Process

As mentioned above, a software can either be verified dynamically or statically. Figure 1.1 visualizes this process in a very simplistic way and has no claim of being complete.

Both unit testing[2] and peer review[3] are processes that depend on a concrete implementation of a system specification, whereas model checking is a more general approach that analyzes the system specification directly. Errors found during model checking imply that there must also be errors in the implementation. On the contrary, if no errors are found in implementation the reverse does not hold. A Reason for this could be that the implementation differs from the system specification under test, e.g. not all properties or requirements are fulfilled by the code. Other reasons could either be that

[1] No piece of software can run without a piece of hardware. Hardware verification is therefore at least as important as software verification. However, we will be concentrating only on software verification.

[2] **Unit Testing:** A dynamic process which tests a concrete implementation with specific test cases.

[3] **Peer Review:** A static process where a group of software engineers analyze the uncompiled code line-by-line without executing it.

the provided unit test cases do not cover all errors in implementation or peer review was not thorough enough.

However, the biggest disadvantage of model checking is its need for resources. In the progress of this paper we will therefore concentrate on possibilities for parallelizing its computation.

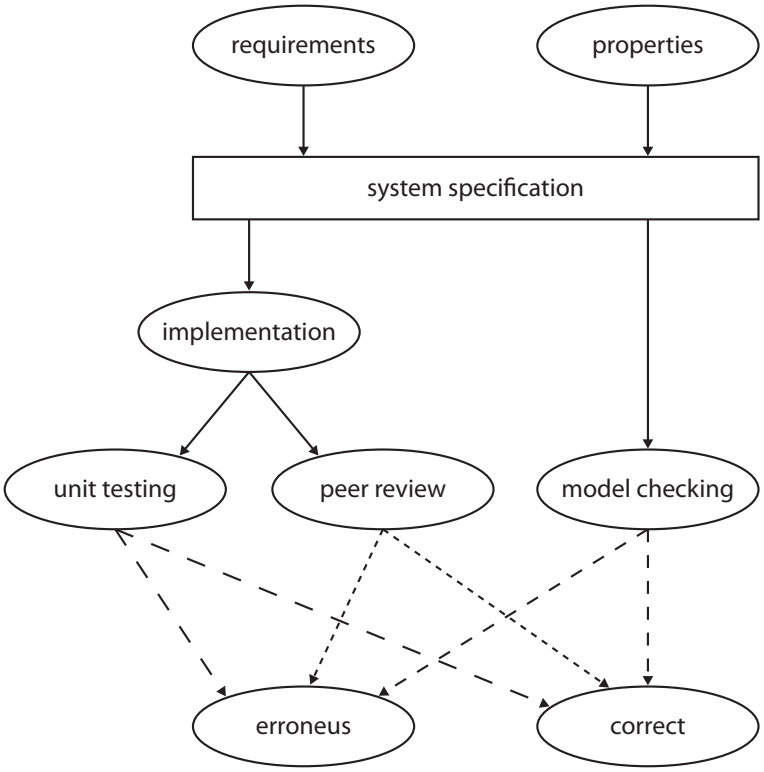

Figure 1.1: Software Verification Scheme

1.2 Comparison of three Verification Techniques

In Table 1.1 three different verification techniques are compared to provide a better overview over software verification. Model checking is a static and formal technique. Therefore, we wanted to compare it with both partially and completely different techniques. Unit testing, being dynamic and functional, denotes a complete different approach, whereas peer review is related to model checking in matters of the verification method.

	Model Checking	Unit Testing	Peer Review
Verification Method	static	dynamic	static
Verification Type	formal	functional	formal or functional
Verification Target	system model	specific implementation	
Memory Costs	⊖ high	relatively low	⊕ none
Computation Costs	⊖ high	relatively low	⊕ none
Time Costs	⊖ high (days to weeks)	relatively low (minutes to hours)	relatively low (hours)
Required Knowledge	⊖ high (to write the input model code for the model checker)	⊕ none to execute tests; average to write tests	⊖ requires good programming skills in order to analyze code
Test Coverage	in more complex systems not the entire model is checked as it would be too expensive; instead, the system model is simplified and only the most interesting/critical parts are checked	during a test run, normally the entire code for which test cases exist is tested	for a specific review a specific code section is specified in advance to be analyzed during a review, i.e. in a review not the entire code is checked ⊖ errors that are spread over different sections might stay undiscovered
Execution Plan	if no errors are found in model, only once	regularly during implementation process ⊖ causes memory, computation and time costs ⊕ ensures that expectations are fulfilled	no code is executed

Table 1.1: Comparison of three verification techniques.

Chapter 2

Theoretical Background

In 1936, *Alan Turing* himself proved the undecidability of the halting problem. Fifteen years later *Henry Gordon Rice* generalized this prove in his doctoral dissertation by stating that it is undecidable to determine whether an algorithm fulfills a non-trivial property. That is to say, in general there is no practicable algorithm to check the correctness of other algorithms.

This chapter should give an understanding of how model checking overcomes this theoretical obstacle by introducing the most basic structures and algorithms which build its foundation. Moreover, abstractions of computer programs and executions, linear temporal logic and its counterpart in computer science, the Büchi automaton, will be defined.

2.1 System Modelling

In theoretical computer science it is common practice to use Turing machines as an abstraction of computer systems. However, as already suggested in the introduction of this chapter, the halting problem and Rice's theorem imply the impracticability of model checking for Turing machines in general.

Simply put, the pitfall of checking for a Turing machine's correctness is its unlimited memory capacity. Because of its endless strip of tape, a Turing machine could have an infinite set of possible states. Checking an infinite set of possible states for correctness is problematic, especially if all states are correct.

Fortunately, in practice there is no computer system with unlimited memory capacity. Therefore, the correctness of a computer program can be checked. However, a replacement for the Turing machine is required to model "real" systems.

2.1.1 Program Graph

A program graph is a mathematical structure that represents a computer program. On a high level of abstraction, a common computer program has a *finite* set of typed variables. Each of those variables have a *finite* domain of possible values. During the execution of the program, actions are taking effect on these variables. The execution of these actions depends on the program's current location and some conditions over the current evaluation of variables.

Definition 1 (Program Graph)**.** A program graph PG over the set of typed variables Var is a 7-tuple $(Loc, Loc_0, G, g_0, Act, \xi, Effect)$ consisting of

- a set of locations Loc,
- a set of initial locations $Loc_0 \subseteq Loc$,
- a set of guards $G \subseteq \mathrm{Cond}(Var)$,
- an initial guard $g_0 \in G$
- a set of actions Act,
- a location transition relation $\xi \subseteq Loc \times G \times Act \times Loc$, and
- an effect function $Effect : Act \times \mathrm{Eval}(Var) \to \mathrm{Eval}(Var)$.

As its name suggests, a program graph could be viewed as a common graph. Its vertices represent a specific location in a program, whereas its edges are guarded actions taking effect on the program's variables (i.e. changes the evaluation of the variables).

Note that a program graph is non-deterministic. The non-determinism could be used to model concurrent computer programs sharing a set of global variables.

2.1.2 Transition System

Until now, a suitable model to abstract a computer program was defined. However, to check the correctness of a computer program it is necessary to analyze the possible executions of that program. Therefore, an abstract structure has to be defined to model the execution paths of a program.

Definition 2 (Transition System)**.** A transition system TS over the set of atomic propositions AP is a 4-tuple (S, S_0, ζ, L) consisting of

- a set of states S,
- a set of initial states $S_0 \subseteq S$,
- a transition relation $\zeta \subseteq S \times S$, and
- a labeling function $L : S \to \mathcal{P}(AP)$.

Analog to a program graph, a transition system can be represented as a graph. Its vertices represent all possible states of a program. The edges represent the possible transitions between these states. The states of the transition system are labeled with so-called atomic propositions. An atomic proposition defines a property that holds in a state.

2.1.3 Transition System Semantics of a Program Graph

To conclude this section, the relation between program graphs and transition systems should be clarified by the definition of its semantics.

Definition 3 (Transition System Semantics of a Program Graph)**.** A transition sytems $TS(PG)$ of a program graph

$$PG = (Loc, Loc_0, G, g_0, Act, \xi, Effect)$$

over the set Var of variables is a 4-tuple (S, S_0, ζ, L) where

- $S = Loc \times \text{Eval}(Var)$,
- $S_0 = \{\langle \ell, \eta \rangle \mid \ell \in Loc_0, \eta \models g_0\}$,
- $\zeta = \{\langle \langle \ell, \eta \rangle, \langle \ell', Effect(\alpha, \eta) \rangle \rangle \mid \langle \ell, g, \alpha, \ell' \rangle \in \xi \wedge \eta \models g\}$
- $L(\langle \ell, \eta \rangle) \mapsto \{a \in AP \mid \eta \models a\}$

Algorithm 1: Conversion of a Program Graph to a Transition System

Input: $PG = (Loc, Loc_0, Act, \xi, \mathit{Effect}),\ AP \subseteq \mathrm{Cond}(\mathit{Var})$
Output: $TS = (S, S_0, \zeta, L)$

$S \leftarrow \varnothing,\ S_0 \leftarrow \varnothing,\ \zeta \leftarrow \varnothing,\ L \leftarrow \varnothing;$
foreach $\eta \in \mathrm{Eval}(\mathit{Var})$ **do**
  if $\eta \models g_0$ **then**
    foreach $\ell_0 \in Loc_0$ **do**
      $S_0 \leftarrow S_0 \cup \{\langle \ell_0, \eta \rangle\};$
      addState(ℓ_0, η);
    end
  end
end
return (S, S_0, ζ, L);

Function addState(ℓ, η)

if $\langle \ell, \eta \rangle \in S$ **then**
  return;
end
$AP' \leftarrow \varnothing;$
foreach $a \in AP$ **do**
  if $\eta \models a$ **then**
    $AP' \leftarrow AP \cup \{a\};$
  end
end
$L(\langle \ell, \eta \rangle) \mapsto AP'$
$S \leftarrow S \cup \{\langle \ell, \eta \rangle\};$
foreach $\langle \ell', g, \alpha, \ell'' \rangle \in \xi$ **do**
  if $\ell' = \ell \wedge \eta \models g$ **then**
    $\eta' \leftarrow \mathit{Effect}(\alpha, \eta);$
    $\zeta \leftarrow \zeta \cup \{(\langle \ell, \eta \rangle, \langle \ell'', \eta' \rangle)\};$
    addState(ℓ'', η');
  end
end

A naive approach to implement an algorithm that converts a program graph to a transition system could look like Algorithm 1.

An obvious yet important observation about the conversion of a program graph is the fact, that it results in a combinatorial explosion of the transition system's states. From this it follows that transition systems representing computer programs of realistic size would exceed the memory capacity of today's computer systems. Therefore, the generation of a whole transition system is not desired. An on the fly approach of the required states is much more suitable.

2.2 Defining Properties

2.2.1 Propositional Logic

One of the simplest ways to define a property of a system are invariants. An invariant is a propositional formula which has to hold in every state of the system. Basically, whether a given transition system fulfills an invariant can be reduced to a graph search algorithm. In Algorithm 2 a possible implementation is presented.

Algorithm 2: Check Transition System for Propositional Formula

Input: $TS = (S, S_0, \zeta, L)$, $\varphi \in \text{Cond}(Var)$

$V \leftarrow \varnothing$; `// the set of visited states`

foreach $s_{0,i} \in S_0$ **do**
  if *check*($s_{0,i}$) = *"error"* **then**
    return *"error"*;
  end
end

return *"ok"* ;

Function check(s)

if $s \in V$ **then**
  return *"ok"*;
end

$V \leftarrow V \cup s$;

foreach $s_i \in \{b \mid \langle a, b\rangle \in \zeta \wedge a = s\}$ **do**
  if $L(s_i) \not\models \varphi \vee$ *check*(s) = "*error*" **then**
    return *"error"*;
  end
end

return *"ok"*;

This algorithm could be extended by a stack that holds the currently checked execution path. In case of a detected error this stack is returned as a counter example to the user.

2.2.2 Linear Temporal Logic (LTL)

In practice, a propositional formula is often not capable of defining the required properties. An extension of propositional logic is required for properties which relate to a behaviour over time. For this purpose, linear temporal logic is introduced.

Syntax

As already mentioned, linear temporal logic is an extension of propositional logic. Therefore, its syntax is extended with two new operators, namely, $\mathcal{U}$ read as *until* and $\bigcirc$ read as *next*.

$$\varphi ::= \text{true} \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathcal{U} \varphi_2$$

Semantics

In contrast to the common logical operators, $\mathcal{U}$ and $\bigcirc$ refer to a temporal behaviour of a system. The formula $\varphi_1 \mathcal{U} \varphi_2$ expresses that φ_1 is true until φ_2 becomes true. On the other hand, $\bigcirc \varphi$ expresses that φ is true in the next step. In addition, two more useful operators are derived from $\mathcal{U}$, namely, $\Diamond$ read as *eventually* and $\Box$ read as *always*.

$$\Diamond \varphi := \text{true} \mathcal{U} \varphi$$
$$\Box \varphi := \neg \Diamond \neg \varphi$$

Linear-temporal Properties

Linear-temporal formulas as introduced above can be used to define a new kind of properties. These properties are called *linear-temporal properties*. There are two main classes of linear-temporal properties. On the one hand, *safety properties* can be used to define situations that *should never* happen. On the other hand, *liveness properties* can be used to define situations that *should* happen at some point in the future.

2.2.3 Automata-Based LTL Model Checking

Previously, linear temporal logic was introduced. It was stated that properties can be defined by using formulas of this logic. However, it was not

clarified how these properties can be checked. There exist dedicated algorithms that could check a computer system for safety and liveness properties. However, we will skip these special cases and concentrate instead on a more general approach. This section introduces the automata-based model checking for linear-temporal formulas.

ω-Regular Properties

So-called ω-regular properties are a superset of linear-temporal properties. These properties can be represented as ω-regular languages. An ω-regular language is a set of infinite words. In automata based model checking a word from such a language is used to describe a possible path through a transition system. The alphabet of these words consists of the possible labels of a transition system states (i.e. the superset of atomic propositions).

Non-Deterministic Büchi Automaton (NBA)

A *non-deterministic Büchi automaton* is an ω-automaton and therefore recognizes ω-regular languages. This makes it a suitable structure to define ω-regular properties.

Definition 4 (Non-Deterministic Büchi Automaton)**.** A non-deterministic Büchi automaton $\mathcal{A}$ is a 5-tuple $(Q, Q_0, F, \Sigma, \delta)$ consisting of

- a finite set of states Q,
- a set of initial states $Q_0 \subseteq Q$,
- a set of accept states $F \subseteq Q$,
- an alphabet Σ, and
- a transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$.

Compared to a non-deterministic finite automaton an NBA differs only in the way it accepts a given word. Instead of accepting a finite word if an accept state is reached at the end of the input, an NBA accepts an infinite word if an accept state is visited infinitely often.

Another noteworthy fact is, that an NBA is more powerful than a deterministic Büchi automaton.

Now, a useful variation of a common NBA is introduced. The so-called generalized non-deterministic Büchi Automaton (GNBA) is the conjunction of multiple NBA.

Definition 5 (Generalized Non-Deterministic Büchi Automaton)**.** A generalized non-deterministic Büchi automaton $\mathcal{G}$ is a 5-tuple $(Q, Q_0, \mathcal{F}, \Sigma, \delta)$ consisting of

- a finite set of states Q,
- a set of initial states $Q_0 \subseteq Q$,
- a set of acceptance sets $\mathcal{F} \subseteq \mathcal{P}(Q)$,
- an alphabet Σ, and
- a transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$.

Compared to a common NBA a GNBA has a set of acceptance sets instead of a single acceptance set. Accordingly, a GNBA accepts an infinite word if its input visits all sets $F_i \in \mathcal{F}$ infinitely often. Note that a GNBA accepts any infinite word if $\mathcal{F}$ is empty.

Construction of an NBA for an LTL Formula

Before introducing the procedure of checking whether a system fulfills a linear-temporal property the construction of an NBA for a linear temporal formula should be introduced.

First of all, let us say φ is an LTL formula. Now, the closure of φ is defined as

$$\text{closure}(\varphi) = \text{subform}(\varphi) \cup \{\neg\psi \mid \psi \in \text{subform}(\varphi)\}$$

where $\text{subform}(\varphi)$ is the set of all subformulas of φ.

In addition, we call a set of subformulas $B \subseteq \text{closure}(\varphi)$ *elementary* if and only if

- B is maximal and consistent with respect to propositional logic, i.e. for $\psi \in \text{closure}(\varphi)$ and $\psi_1 \wedge \psi_2 \in \text{closure}(\varphi)$:
  - $\psi \notin B \Leftrightarrow \neg\psi \in B$
  - $\psi_1 \wedge \psi_2 \in B \Leftrightarrow \psi_1 \in B \wedge \psi_2 \in B$
  - $\text{true} \in \text{closure}(\varphi) \Rightarrow \text{true} \in B$
- B is locally consistent with respect to $\mathcal{U}$, i.e. for $\psi_1 \,\mathcal{U}\, \psi_2 \in \text{closure}(\varphi)$:
  - $\psi_1 \,\mathcal{U}\, \psi_2 \in B \wedge \psi_2 \notin B \Rightarrow \psi_1 \in B$
  - $\psi_2 \in B \Rightarrow \psi_1 \,\mathcal{U}\, \psi_2 \in B$

Based on this foundation, the conversion of a GNBA to an LTL formula is defined as follows.

Definition 6 (GNBA for LTL Formula)**.** A generalized non-deterministic Büchi automaton $\mathcal{G}(\varphi)$ of an LTL formula φ is a 5-tuple $(Q, Q_0, \mathcal{F}, \Sigma, \delta)$ consisting of

- a finite set of states $Q = \{B \subseteq \text{closure}(\varphi) \mid B \textit{ is elementary}\}$,
- a set of initial states $Q_0 = \{B \in Q \mid \varphi \in B\}$
- a set of acceptance sets $\mathcal{F} = \{F_{\psi_1 \mathcal{U} \psi_2} \mid \psi_1 \mathcal{U} \psi_2 \in \text{closure}(\varphi)\}$ where $F_{\psi_1 \mathcal{U} \psi_2} = \{B \in Q \mid \psi_1 \mathcal{U} \psi_2 \notin B \vee \psi_2 \in B\}$
- an alphabet $\Sigma \subseteq \mathcal{P}(AP)$, and
- a transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ for $B \in Q$ and $A \in \Sigma$:
  - $A \neq B \cap AP \Rightarrow \delta(B, A) \mapsto \varnothing$
  - $A = B \cap AP \Rightarrow \delta(B, A) \mapsto \{B' \in Q\}$ where
    - $\bigcirc \psi \in B \Leftrightarrow \psi \in B'$
    - $\psi_1 \mathcal{U} \psi_2 \in B \Leftrightarrow (\psi_2 \in B) \vee (\psi_1 \in B \wedge \psi_1 \mathcal{U} \psi_2 \in B')$

Now, the resulting GNBA can be converted to a common NBA. This NBA could now be combined with the transition system under test.

Definition 7 (Product of a Transition System and an NBA)**.** The product $TS \otimes \mathcal{A}$ of a transition system $TS = (S, S_0, \zeta, L)$ and an NBA $\mathcal{A} = (Q, Q_0, F, \Sigma, \delta)$ is a transition system $(S', S_0{}', \zeta', L')$ over $AP' = Q$ consisting of

- a set of states $S' = S \times Q$,
- a set of initial states $S_0' = \{\langle s_0, q\rangle \mid s_0 \in S_0 \wedge \exists q_0 \in Q_0 . \delta(L(s_0)) = q_0\}$,
- a transition relation $\zeta'(\langle s, q\rangle) = \{\langle s', q'\rangle \mid \langle s, s'\rangle \in \zeta \wedge \delta(q, L(s')) = q'\}$, and
- a labeling function $L' : S \times Q \to \mathcal{P}(Q)$, where $L'(\langle s, q\rangle) = \{q\}$.

Finally, the resulting transition system can be tested for correctness by searching for a cycle that contains a state labeled with an accept state of the NBA.

Chapter 3

Parallelization of Model Checking

In the previous chapter we discovered that the conversion of a program graph to a transition system suffers from a combinatorial explosion of the transition system's states. To avoid the problem of memory exhaustion, an on-the-fly approach to generate the transition system was suggested. But even with this approach a good portion of the transition system may be generated. This is especially problematic for an execution on graphics cards, given the fact that even high-end devices come with only approximately two gigabytes of memory. Moreover, holding a single set of visited states is problematic if it comes to a possible parallelization. As soon as more than one thread accesses this set, they have to be synchronized. This could reduce the performance enormously.

Therefore, the naive approach of "simply" visiting all newly generated states concurrently might result in a worse performance compared to a conventional single threaded execution. Alternatively, traversing the transition system in a depth-first search manner and keeping only the states that belong to the currently checked execution path, the memory usage could be reduced to a practicable level and the required sets of visited states could be split per thread. However, this may cause the generation of some states that were already checked in another execution path and therefore reduce the performance as well.

The impact of this performance loss could be reduced by partitioning the transition system into smaller subgraphs which could be processed independently. If there was an efficient way of finding independent subgraphs such that each could be treated separately in an own thread, this could mean a significant improvement in time needed for model checking. Obviously, this would require an analysis of the whole transition system. Therefore, an analysis of independent subgraphs should happen before it is converted to a transition system.

3.1 By Analyzing a Program Graph's Variables

In this section we will concentrate on the attempts made trying to find the before mentioned independent subgraphs of a program graph by analyzing its variables. We will not only cover the working but also the failed attempts. In Section 3.1.1 an illustrative example is introduced on which the subsequent sections rely on. Section 3.1.2 will discuss some failed attempts and finally Section 3.1.3 will cover a working algorithm.

As already introduced in Chapter 2, a program graph not only consists of locations and actions, but also of guards which must hold in order that an action can take place. In the course of this section we will be using the term *transition* of a program graph instead of bothering the reader with differentiations between actions, guards and locations. A transition will denote a change from one location to another by executing an action whose guards are satisfied. Moreover, to avoid unnecessary complexity, we will leave out any guard handling. Instead, we will define a transition to depend on a variable if the guards and/or actions rely on it.

Definition 8 (Transition)**.** A transition denotes a change from one location to another by executing an action whose guards are satisfied. Moreover, a transition depends on a variable if the guards and/or actions rely on it.

Definition 9 (Independent Subgraph)**.** A subgraph s_1 is independent if the transitions in s_1 do not rely on transitions in any other subgraph s_2, where s_1 and s_2 are not the same.

3.1.1 Introductory Example

To illustrate the algorithms, in this section we will be working with the program graph shown in Figure 3.1. For clearer referencing, each transition is labelled with $\alpha_1..\alpha_9$. Furthermore, the program graph can be represented as a matrix like in Figure 3.2 (page 19), where the rows stand for transitions and the columns stand for the available variables in the program graph. Each cell in the matrix contains either the value 0 or 1 depending on whether a transition uses a variable (1) or not (0).

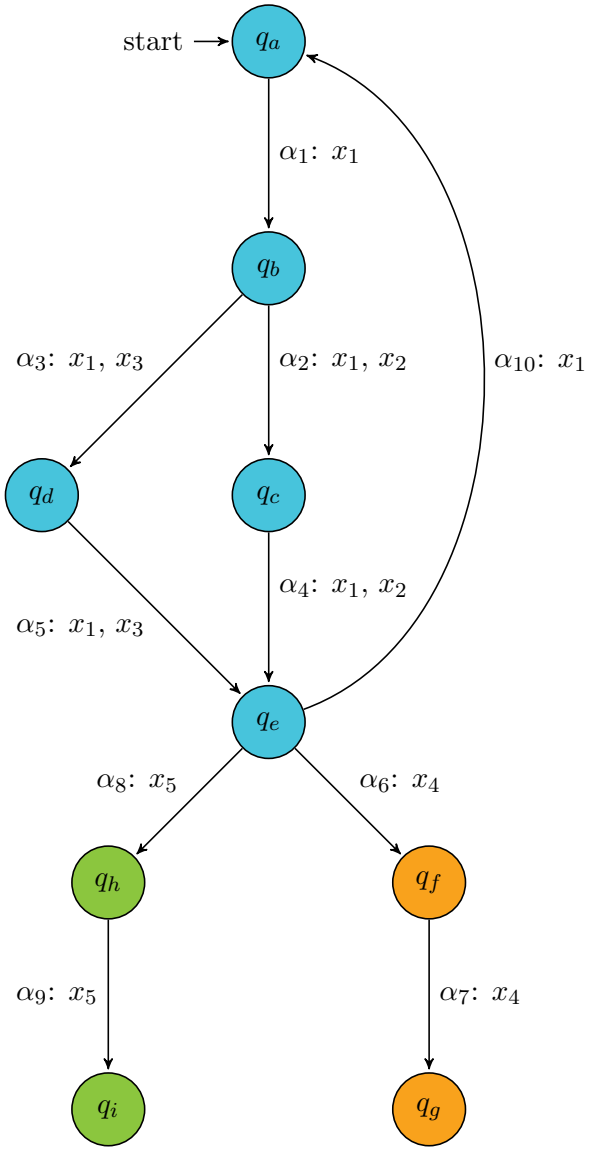

Figure 3.1: Exemplary Program Graph where the colored nodes represent independent subgraphs.

3.1.2 Matrix Approach with Gate Logic

In this section we will work with the matrix representation of the program graph. The aim of this approach is to find independent subgraphs of the program graph by using simple matrix operations and logic gates such as XOR, NAND, etc. Specially when executed on a graphic card, a working algorithm with an approach like this would be very efficient.

$$m_{pg} = \begin{array}{c} \\ \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \alpha_5 \\ \alpha_6 \\ \alpha_7 \\ \alpha_8 \\ \alpha_9 \\ \alpha_{10} \end{array} \begin{array}{c} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 3.2: Matrix Representation of exemplary Program Graph.

The naive idea to find related variables was to compose a *gate matrix*, i.e. a matrix that followed a specific pattern and which then would be concatenated with the program graph matrix m_{pg} by a gate operation. The gate matrix m_g is built by an algorithm similar to the one described in Algorithm 3.

Algorithm 3: Build the gate matrix

```
Input: m_pg ;                                  // the program graph matrix
Output: m_g ;                                  // the gate matrix
foreach row r in m_pg do
    foreach column c in m_pg do
        if m_pg[r][c] == 1 then
            set entire row c in m_g = 1
        else
            if m_g[r][c] != 1 && m_g[r-1][c] == 0 then
                m_g[r][c] = 0
            end
        end
    end
end
```

The gate matrix m_g based on Algorithm 3 with matrix m_{pg} as input would look like:

$$m_g = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 3.3: Gate matrix

However, once we had studied the gate logic a bit further, it was quite obvious that such an approach could not work. First, we were not able to find a universally working algorithm that could build up the gate matrix in a way that would be useful for further processing by a gate logic operation. Furthermore, with the examined logic gates (AND, OR, XOR, NAND) it would not be possible to differentiate between multiple independent subgraphs as the logic would be applied on all variables and the dependencies would all accumulate to one big heap (as shown in Figure 3.4). Also the gate matrix would not be unambiguous as different ways of ordering the transitions in matrix m_{pg} would lead to completely different gate matrices m_g. Obviously, this would have a direct impact on the dependencies between transitions.

$$m_{pg} \oplus m_g = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \oplus \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 3.4: Example with XOR operation

3.1.3 Iterative Approach

The iterative approach consists of repetitive merging of arrays $a_1..a_n$ where 1..n characterizes a variable in the program graph matrix. In this way, each of these arrays denote dependent transitions regarding one specific variable. Having this information, the next step is to find dependent transitions with reference to all variables of the program graph. This is where the iterative algorithm comes into action. Certainly it is not the most efficient one and it has potential for improvement. However, it will meet our purposes.

In the first step, the program graph matrix must be divided into multiple arrays as described above. These arrays will then be needed in Step 2 for the iteration and merge process.

Algorithm 4: Iterative algorithm: Step 1

Input: m_{pg} ; `// copy of the program graph matrix`

foreach *variable v in* m_{pg} **do**
  create array a_n containing the transitions influenced by v
end

Now we have multiple arrays which are much easier to handle than a matrix. As it is essential that already merged arrays are not merged over again, using arrays is the preferred way to go for us, as in a matrix this would be much harder to implement and keep track of.

In Step 2 we will need another set of arrays which we well call *merge arrays*. For better differentiation from the arrays in Step 1, in addition to

the integer index, these arrays will be labelled with an m, i.e. there will be merge arrays $a_{m1}..a_{mn}$.

However, because a merge array a_{mn} is modified during runtime, it must be assured by the implementation, that every element in the array is handled by the foreach-loop.

Algorithm 5: Iterative algorithm: Step 2

```
a_mn ← a_1 ;                                   // temporary merge array
L1: foreach e ∈ a_mn do
    if e ∈ any other array a_x then
        a_mn = merge(a_mn, a_x)
        free(a_x) ;                  // convenience step to free memory
    end
end
if ∃ unmerged unhandled array a_x then
    create another temporary merge array a_mn
    goto L1 and try to merge a_mn with any left array a_x
end
```

3.2 By Analyzing a Program Graph's Structure

In this section we introduce another approach to find independent subgraphs of a program graph. In contrast to the previous section, the variables are completely ignored. Instead, the structure of a program graph is analyzed and subgraphs are defined as strongly connected components.

3.2.1 Irreversible Transition

We observed that transitions between strongly connected components have an interesting property when it comes to a possible partitioning of a program graph. We call these transitions *irreversible*.

Definition 10 (Irreversible Transition)**.** A transition $\langle \ell, g, \alpha, \ell' \rangle \in \xi$ in a program graph $PG = (Loc, Loc_0, G, g_0, Act, \xi, Effect)$ is irreversible if ℓ is not reachable from ℓ'.

In Figure 3.5 an exemplary program graph is depicted. Nodes of the same color form strongly connected components. Dashed edges represent irreversible transitions.

The preceding locations are irrelevant after passing an irreversible transition. Therefore, every irreversible transition splits the program graph in two independent subgraphs. Based on the relation of transition systems and

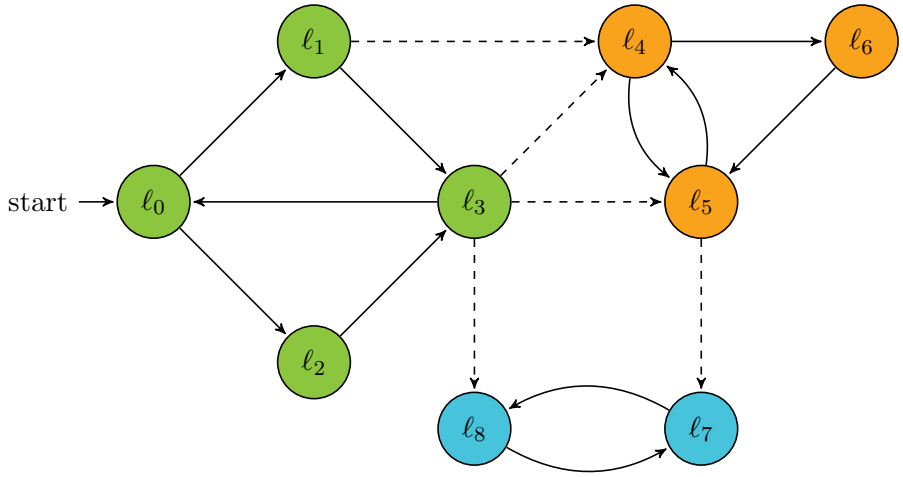

Figure 3.5: Exemplary Program Graph where the colored nodes represent independent subgraphs and dashed edges represent irreversible transitions.

program graphs that follows directly from Definition 3 (page 9), we can apply this observation on the derived transition system. For convenience, we recall the relevant parts of this definition.

- $S = \mathit{Loc} \times \mathrm{Eval}(\mathit{Var})$
- $\zeta = \{\langle\langle \ell, \eta\rangle, \langle \ell', \mathit{Effect}(\alpha, \eta)\rangle\rangle \mid \langle \ell, g, \alpha, \ell'\rangle \in \xi \wedge \eta \models g\}$

Let us say there is a state s in the transition system which is derived from a location ℓ in the program graph. Furthermore, there is a state s' which is derived from a location ℓ' and s is reachable from s'. The relation above now implies that there cannot exist such a state s if ℓ cannot be reached from ℓ'.

This means that every transition in the generated transition system that is derived from an irreversible transition in the program graph is irreversible itself. For the purpose of parallelization, every time an irreversible transition is passed, a new thread can handle the following subgraph on its own without needing synchronization.

Basically, to identify strongly connected components a depth-first search can be used. For a massive parallelization, it is not as suitable as a breadth-first search. Nevertheless, this fact is of no importance because the proposed analysis is based on the relatively small program graph and could therefore be executed prior to the verification phase on a CPU without noteworthy performance loss.

While studying the impact of irreversible transitions we assumed that this approach can be applied to the generated NBA to obtain an even higher partitioning of the resulting transition system. Though, we realized that the

conversion of an LTL formula to an NBA results in an NBA with exponential size compared to the length of the LTL formula. Hence, an analysis of the NBA seems to be impracticable, but a further analysis of the LTL might still reveal potential for parallelization.

3.3 By the Linear Map Approach

This approach arose from the idea that an action in the program graph can be reduced to a multiplication and an addition of the evaluation of the variables.

$$\alpha := \begin{cases} v_1' := \alpha_{1,1}v_1 + \alpha_{1,2}v_2 + \alpha_{1,3}v_3 \\ v_2' := \alpha_{2,1}v_1 + \alpha_{2,2}v_2 + \alpha_{2,3}v_3 \\ v_3' := \alpha_{3,1}v_1 + \alpha_{3,2}v_2 + \alpha_{3,3}v_3 \end{cases}$$

This would mean that an action α can be represented by a transformation matrix M_α while the variables are represented as a vector v.

$$M_\alpha = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}, v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

In addition, a guard may be represented by a conjunction of multiple inequations. The conjunction of conditions of a guard defines an intersection of areas in a hyperspace.

$$g := \begin{cases} & 0 \leq g_{1,1}v_1 + g_{1,2}v_2 + g_{1,3}v_3 + g_{1,4} \\ \wedge & 0 \leq g_{2,1}v_1 + g_{2,2}v_2 + g_{2,3}v_3 + g_{2,4} \\ \vdots & \\ \wedge & 0 \leq g_{n-1,1}v_1 + g_{n-1,2}v_2 + g_{n-1,3}v_3 + g_{n-1,4} \\ \wedge & 0 \leq g_{n,1}v_1 + g_{n,2}v_2 + g_{n,3}v_3 + g_{n,4} \end{cases}$$

This intersection may allow its representation as a simplex. If it is possible to determine the corners of this simplex, these could be mapped to their new coordinates in the hyperspace by multiplying them with the corresponding transformation matrix.

On the one hand, the resulting subspace may be checked for intersection with the space defined by the given properties. On the other hand, this may lead to a possible determination of cycles in the program graph, without the conversion to a transition system. To visualize this approach a two-dimensional example is given in Figure 3.6.

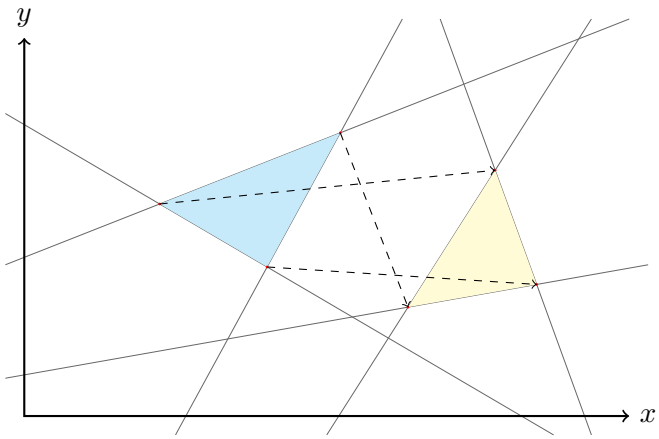

Figure 3.6: Example for Linear Map of a Guarded Area.

Because of the amount of unfounded assumptions and our limited knowledge on linear algebra we decided not to pursue this approach further.

Chapter 4

Conclusion

To close this seminar paper we will conclude our findings. In our opinion, there certainly exists potential for parallelization. However, the time given to finish this seminar paper was limited. For further analysis of the parallelization possibilities more time would be required in order to make ourselves even more familiar with the smallest details of model checking.

On the one hand, the problems to be solved belong to the most complex ones, i.e. in computational complexity theory you move in the field of NP-hard problems. In order to handle this complexity some processes in model checking might have to be simplified in such a way that they become efficiently solvable by a parallel algorithm.

On the other hand, if you want to parallelize as many elements of a parallelized algorithm as possible, such an algorithm would need to be implemented partially or even completely from scratch. After having looked into the source code of SPIN we think that its structure is too complex to be useful for a parallelized adaptation, not least, because the source code is quite old.

In a next step, we think that other elements than the program graph should be analyzed. As already mentioned in Chapter 3, e.g. the transformation from LTL to GNBA could be a next point of action.

However, one thing to bear in mind is that although graphic cards could provide an enormous performance improvement, they have two significant disadvantages. They only have a very limited amount of memory - usually one or two gigabytes. For this reason, the performance gain can additionally be defeated if there are too many copy transactions between main memory and the graphic card. These components are only connected by a PCI bus which is very slow compared to the computation power of the graphic card itself.

List of Figures

List of Tables

Bibliography

[1] Spin. http://spinroot.com.

[2] Jens Marco Bendisposto. Integration of the prob model checker into eclipse. Düsseldorf, Germany, 2006.

[3] Baier Christel and Katoen Joost-Pieter. *Principles of Model Checking*. The MIT Press, Cambridge Massachusetts and London England, 2008.

[4] Klaus Havelund and Jens Ulrik Skakkebaek. Applying model checking in java verification. Moffet Field CA USA and Stanford CA USA, 1999.

[5] Gerard J. Holzmann and Dragan Bošnački. The design of a multi-core extension of the SPIN model checker, 2007.

[6] Aditya Kanade. Automated verification. http://drona.csa.iisc.ernet.in/~kanade/teaching/2012/E0223/, 2012. Last visited on 13th December 2012.